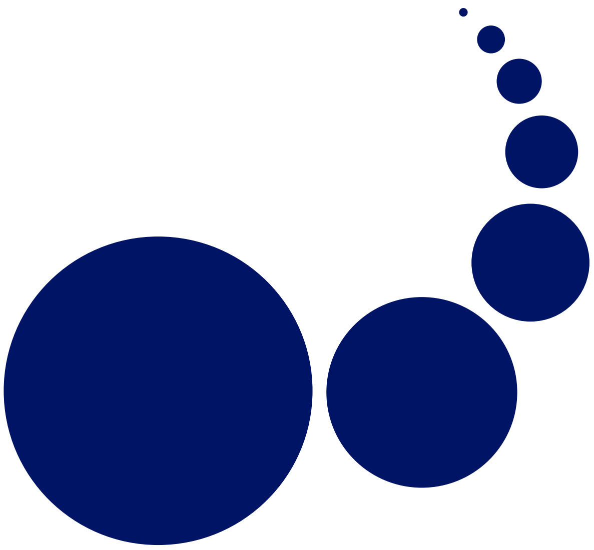
SCALABLE COMPUTING
Practice and Experience

Special Issue: Recent Developments in Multi-Core Computing Systems

Editors: Sabri Pllana, Siegfried Benkner



Volume 9, Number 3, September 2008

ISSN 1895-1767

UNIVERSITÄT SALZBURG

UNIVERSITATEA DE VEST DIN TIMISOARA

SUBSCRIPTION INFORMATION: please visit http://www.scpe.org

Scalable Computing: Practice and Experience

Volume 9, Number 3, September 2008

TABLE OF CONTENTS

Scalable Computing: Practice and Experience
Volume 9, Number 3, p. i. http://www.scpe.org

ISSN 1895-1767
© 2008 SCPE

INTRODUCTION TO THE SPECIAL ISSUE: RECENT DEVELOPMENTS IN MULTI-CORE COMPUTING SYSTEMS

The improvement of processor performance by increasing the clock rate has reached its technological limits. As a result, new processor generations rely on multiple processor cores on a chip in order to increase performance, while mitigating problems like energy consumption, heat dissipation and design complexity. We are now witnessing the emergence of multi-core processors in all markets from laptops and game consoles to servers and supercomputers. However, exploiting the full potential of multi-core computing systems for application programs is a highly complex task posing many open research questions.

The 2008 International Workshop on Multi-Core Computing Systems (MuCoCoS'08) was organized in Barcelona, Spain, March 7th. In total 20 papers were submitted to MuCoCoS'08, of which 10 papers were selected by the program committee for presentation. Each paper was thoroughly evaluated by members of the program committee based on the significance to the CFP, originality, and contribution to theory/practice. From the 10 papers presented at MuCoCoS'08 we have selected six papers for this special issue. The selected papers address a representative set of issues related to the recent developments in multi-core computing systems.

In the first paper—*"An Asynchronous API for Numerical Linear Algebra"* by A. R. Brodtkorb—an approach for general purpose computation on Graphics Processing Units (GPUs) using MATLAB is described. The applicability of the approach is demonstrated for several linear algebra algorithms.

The second paper—*"On the Potential of NoC Virtualization for Multicore Chips"* by J. Flich, S. Rodrigo, J. Duato, T. Sødring, Å. G. Solheim, T. Skeie and O. Lysne—is a position paper that provides arguments in favor of Network-on-a-Chip (NoC) virtualization. It is argued that NoC virtualization, associated with topology agnostic routing, minimizes fragmentation, provides performance isolation, facilitates cache coherency, helps dealing with manufacturing defects, and makes it easy to turn off unused cores for energy saving. These arguments are supported by a series of simulation results.

In the third paper—*"Single-pass List Partitioning"* by L. Frias, J. Singler and P. Sanders—a new algorithm to partition a list into pieces is described, which is potentially useful for parallelizing problems on multi-core processors. The approach described in this paper requires a single pass over the list using sub-linear storage to perform the partitioning, while previous approaches require two passes. Authors evaluate their approach empirically using an AMD Opteron based system.

The fourth paper—*"Efficient Implementation of WiMAX Physical Layer on Multi-core Architectures with Dynamically Reconfigurable Processors"* by W. Han, Y. Yi, M. Muir, I. Nousias, T. Arslan and A. T. Edorgan—proposes the use of dynamically reconfigurable processors for wireless applications. A simulator for multi-core systems is used for the evaluation of the proposed approach.

In the fifth paper—*"Latency Impact on Spin-lock Algorithms for Modern Shared Memory Multiprocessors* by J. C. Meyer and A. C. Elster—an empirical evaluation of spin-lock algorithms on IBM System p575+ and SGI Origin 3800 computing systems is presented.

The last paper—*"Parallel Advanced Video Coding: Motion Estimation on Multi-cores"* by S. Momcilovic and L. Sousa—presents a motion estimation technique for multimedia processing, and its parallelization on the Cell Broadband Engine processor. The authors report very good performance in terms of frame rate on the Cell processor. The paper presents details on work partitioning among the PPU and SPE's that may be applicable to other applications on Cell.

We are grateful to all authors for submitting their papers to MuCoCoS'08, to program committee members for their efforts to evaluate papers within a short period, and to David A. Bader for delivering the keynote speech. Furthermore, we would like to acknowledge the support of Marcin Paprzycki (SCPE editor-in-chief) and Alexander Denisjuk (SCPE technical and managing editor).

Special Issue Editors
Sabri Pllana and Siegfried Benkner

Scalable Computing: Practice and Experience
Volume 9, Number 3, pp. 153–163. http://www.scpe.org
ISSN 1895-1767
© 2008 SCPE

AN ASYNCHRONOUS API FOR NUMERICAL LINEAR ALGEBRA

ANDRÉ RIGLAND BRODTKORB*

Abstract. We present a task-parallel asynchronous API for numerical linear algebra that utilizes multiple CPUs, multiple GPUs, or a combination of both. Furthermore, we present a wrapper of this interface for use in MATLAB. Our API imposes only small overheads, scales perfectly to two processor cores, and shows even better performance when utilizing computational resources on the GPU.

Key words: asynchronous, multicore, GPU, MATLAB, CUBLAS, double precision

1. Introduction.

Algorithms from numerical linear algebra are important tools with a variety of uses. Common to many of these algorithms is that they require a substantial amount of computation time. Therefore, there has been a lot of research into developing optimized APIs and libraries such as BLAS [14], FLAME [15], LAPACK [2] and PLASMA [13]. However, when these libraries are used, they stall the program execution until each algorithm has completed. This prevents overlapping heavy computation with other operations, such as reading in new data from disk, without complex multithreaded programming. We present an interface that offers asynchronous and task-parallel execution of BLAS functions on different *backends*. This enables easy overlap with other operations and enables us to utilize several processor cores, multiple graphics processing units (GPUs), or both in combination.

The interface we present consists of a frontend that exposes familiar BLAS functions and several backends that implement them. The frontend schedules execution of the algorithms to the different backends run-time based on a set of simple criteria to utilize all available processing power. We further present a wrapper of this interface for use in MATLAB. This is a natural extension to our previous work [8], where we showed speed-ups using the GPU as a computational resource in MATLAB.

We have focused on utilizing multicore CPUs in conjunction with one or more GPUs. However, our ideas could also have been applied to other accelerator cores such as the Cell broadband engine (Cell BE) [16] or field programmable gate arrays (FPGAs) [10].

The rest of this article is sectioned as follows: Section 2 discusses current multi-core hardware and software trends, and we relate our API to existing programming languages and APIs. Readers familiar with these trends might opt to jump straight into Section 3, where we contrast our contribution to related work. Section 4 describes our interface, followed by Section 5 where we show performance results. We end this article with some concluding remarks and future research directions in Section 6.

2. Current Trends in Parallel Commodity Hardware.

The achieved performance of computer programs has traditionally, with only minor adjustments, increased with new hardware generations. The performance gain has mainly come from an increase in four metrics: size of system memory, processor core clock frequency, speed of system memory, and number of instructions per clock cycle. The size of system memory is limited to 4 GiB on 32 bit systems, but this limitation was increased to a theoretical 256 GiB for 64 bit systems with the emergence of the x86-64 instruction set in 2003. The increase of the three other metrics, however, has come to a halt. The factors limiting further growth have been called the *power wall*, *memory wall*, and *ILP wall*, constituting a “brick wall for serial performance“ [3].

The power wall prevents further increases in clock frequency. Increasing the clock frequency requires reducing the gate size and increasing the supply voltage. Both reducing the gate size and increasing the supply voltage leads to an increase in leakage power and generated heat. This generated heat is proportional to the square of the frequency, and we seem to have reached a physical limit to what the chips can withstand without exotic cooling.

The size of on-chip caches has increased drastically in recent years. This is because the bandwidth to off-chip memory has become relatively slower and slower, referred to as the *von Neumann bottleneck*. One of the reasons for this slow-down is that there is a physical limit to the number of pins connecting the processor to the motherboard and main memory. The speed per pin is also limited, further obstructing speed increases. This limitation is commonly known as the memory wall.

*SINTEF, Dept. Appl. Math., P.O. Box 124, Blindern, N-0314 Oslo, Norway.

The third wall is referred to as the ILP wall. Increasing the number of instructions executed per clock cycle is a result of instruction level parallelism (ILP). Complex logic executes dependent instructions concurrently by predicting the program flow. Further development of ILP, however, is currently unfeasible because of the exponentially difficult task of predicting future instructions.

These three walls mark the end of serial performance increase for the time being, and the era of multicore computing has begun. Dual- and quad-core processors have already become the mainstream of processors, and we will see a steady increase in the number of cores in the future. This represents a great challenge to computer scientists and algorithm designers as most algorithms are designed for serial architectures, thus unable to benefit from these processor designs.

In addition to the processing power offered by the multicore CPU, most modern computers are also equipped with one or more dedicated graphics cards to accelerate rendering in games. The graphics card looks a lot like a separate computer by itself: it contains a GPU, dedicated graphics memory, and a graphics BIOS. The cards are accessed through the graphics driver that offers one or more graphics APIs.

In current games, you typically have a screen resolution of 1600×1200 pixels, with at least 60 frames drawn each second. This totals to over one hundred million pixels each second. The number of operations per pixels can be very large, as modern games often use advanced rendering techniques requiring several rendering passes per frame. To meet these demands graphics cards have developed an extreme computational capacity. Current consumer level GPUs can have a theoretical peak performance of over 1 TFLOPS, compared to CPUs that have less than 150 GFLOPS. The GPU also outperforms the CPU when it comes to memory bandwidth. Motherboards give the CPU access to main memory at speeds up to 25 GiB/s, while the typical memory speed of a commodity-level computer is around 13 GiB/s, or even less. For GPUs, however, the memory speeds reach over 140 GiB/s. And with good reason. It takes a lot of bandwidth to process over one hundred million pixels each second.

Using graphics processors to accelerate non-graphical applications is a field that has gained in popularity, see e.g., `www.gpgpu.org` and Owens et. al. [25]. One of the main bottlenecks with such use of the GPU is the relatively slow data bus between GPU and main memory. Heterogeneous processor designs such as the Cell BE and the coming AMD Fusion processors remove this bottleneck by incorporating accelerator cores on the same silicon die as the CPU. It is commonly accepted that using such accelerator cores outperform pure CPU implementations for a variety of processing tasks. See for instance Brodtkorb et. al. [9] for an example of a comparison between multi-core CPUs, the GPU, and the Cell BE.

The supercomputer community is also using accelerator cores to achieve higher performance. Bull have disclosed that they will build a supercomputer with 1080 octa-core Nehalem CPUs in conjunction with 96 NVIDIA Tesla [17] GPUs for Grand Equipement National de Calcul Intensif in France. The RoadRunner project at Los Alamos National Laboratory utilizes 6,948 AMD dual-core CPUs in conjunction with 12,960 Cell BE processors, and is the first machine to achieve over one PFLOPS sustained performance. It is currently the most powerful supercomputer in the world, and also one of the most energy efficient.

2.1. Programming languages and APIs. There are already many different programming languages and APIs for multicore computing, such as POSIX Threads (pthreads) [12], Intel Threading Building Blocks (TBB) [27] and OpenMP [24]. All these libraries use threads to utilize multiple processor cores.

POSIX Threads is a low-level API where the programmer handles each thread explicitly. This level of access offers great flexibility, but at the same time requires a lot from the programmer. Programs written using pthreads are subject to thread issues such as *race-conditions*, *deadlocks*, *starvation*, and *priority failures*, all of which require the programmer to create and maintain complex logic.

TBB is a C++ library that uses threads, but focuses on *tasks*. Given a parallel algorithm, TBB aids in partitioning and scheduling the tasks, thus reducing the risk of thread issues. It also implements task stealing that enables dynamic load balancing. The task stealing is performed runtime by splitting large tasks into smaller tasks, and then redistributing them to idle threads. Starting these tasks is 100 times faster than starting a thread on Windows XP.

OpenMP consists of a set of compiler commands. The programmer outlines the parallel sections of the code using a set of pragmas, guiding the compiler where to perform parallelization. The compiler recognizes these pragmas, and generates code that can execute in parallel on a shared memory machine. OpenMP is used in the OpenMP Multi-Threaded Template Library, which is a parallel implementation of `<numeric>` and `<algorithm>` from the standard template library (STL).

The asynchronous linear algebra API we present here uses Boost::Threads, a portable API for threading that is slightly higher level than POSIX Threads. We offer a task-parallel API, using ideas similar to TBB. However, we do not implement task subdivision and task stealing, and focus only on mathematical operations defined in the BLAS API.

Traditionally, the GPU had to be accessed through a graphics API such as OpenGL [28] or DirectX [20]. Accelerating computations using the GPU thus required that the algorithm could be rewritten in terms of operations on graphical primitives, a tedious and error prone solution for non-graphics use. An increasing demand for non-graphical APIs sparked a lot of research into creating other abstractions, and there are currently two dominant APIs for *stream computing*; NVIDIA CUDA [23] and Brook [11].

CUDA is a data-parallel programming language and paradigm that requires algorithms to be structured into blocks of independent computation. It can execute on recent NVIDIA GPUs, or in emulation mode on the CPU. The CUDA Zone at the NVIDIA website contains an updated list of academic and industrial uses of CUDA. This programming paradigm has also been implemented for multicore CPUs as MCUDA [29].

Brook is an open source API developed at Stanford. The Stanford version has not reached a final release, but has spurious updates. AMD has developed an extension to the API called Brook+ [1] for their GPUs, but the language specification is not fully implemented. The Brook API is used by Folding@home project to simulate protein folding on both GPUs and the Cell BE in the PlayStation 3 gaming console.

PyStream is a python interface to CUDA, CUBLAS [22] and CUFFT [21] that supports seamless data-transfer between CPU and GPU memory space. TechX has stopped developing this open source version, but continues developing GPULib, a library that accelerates mathematical functions using the GPU. GPULib offers bindings to Python, MATLAB and the Interactive Data Language (IDL).

RapidMind [19] is a high-level stream programming API with backends for both multicore CPUs, GPUs, and the Cell BE. The backends are themselves responsible for low-level optimization, and RapidMind has published results where highly optimized RapidMind code outperformed hand-tuned code [26].

Our approach is similar to GPULib, as we also offer transparent data-transfers and asynchronous execution of mathematical functions on the GPU. As with RapidMind, we also support different backends for our interface. RapidMind offers a general programming framework for data-parallel execution, whereas we focus on task-parallel mathematical functions. In addition, we support using multiple backends simultaneously for task-parallel execution.

3. Related Work.

For an overview of history and recent developments in linear algebra on the GPU, Owens et. al. [25], and the references therein, provide a good summary. We have previously presented an interface between MATLAB and the GPU [8], where algorithms from numerical linear algebra were accelerated by the GPU. This early work required the algorithms to be written in terms of operations on graphical primitives, as only graphics APIs were available for our NVIDIA GPU. Accelereyes are developing a product called Jacket to accelerate MATLAB using the GPU. They use ideas very similar to our original paper but also offer OpenGL visualization.

Faticia and Jeong [18] have also presented a coupling of MATLAB and the GPU. They used CUFFT to accelerate simulations run from MATLAB, and reported the GPU version to be 14 times faster for a 2D isotropic turbulence problem. GLAME@LAB [6] uses CUBLAS to accelerate FLAME@LAB [7], the MATLAB interface to the FLAME API [15]. They showed speedups over Octave for several different algorithms. The performance of CUBLAS has been explored by Barrachina et. al. [5]. They also presented a hybrid sgemm (**s**ingle-precision **ge**neral **m**atrix **m**ultiply) algorithm that utilized both CPU and GPU computational power by splitting the computation into blocks.

Our contribution offers asynchronous execution of mathematical functions that enable us to overlap computation with other operations. We also offer asynchronous data transfer, and task-parallel execution of BLAS functions, being able to benefit from both multiple CPUs, multiple GPUs, or a mixture of both.

4. Interface.

We present an asynchronous interface to algorithms in numerical linear algebra. The interface is divided up into a frontend exposed to the user, and one or more backends. The backends can be implemented for different processors, allowing for a heterogeneous processing platform. This section describes the frontend, the backends, and a MATLAB wrapper of the frontend. The interface is general enough to support new backends, e.g., an FPGA backend, and new frontend wrappers, e.g., a Python wrapper.

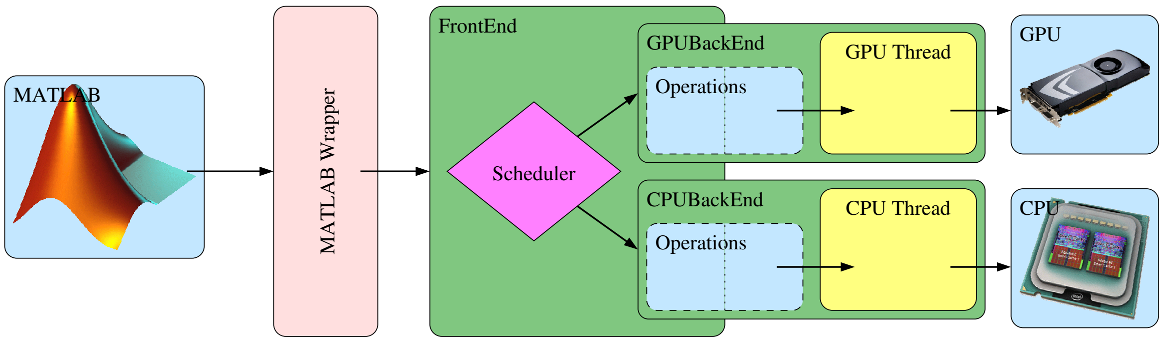


Fig. 4.1. *The interface design.*

Figure 4.1 shows the main layout of the interface. The frontend is responsible for creating *tasks*, and scheduling the tasks to the backends. These tasks are lightweight data-structures that transport pointers to input and output data between the frontend and the backends. The scheduling algorithm tries to find the best backend to schedule the process to, and then adds the task to the queue of the appropriate backend. The backends run asynchronously in separate threads, continuously processing tasks from their task-queue.

4.1. Frontend. The frontend defines functions exposed by our asynchronous API. It is the entry-point for a wrapper or user, and contains a few data-handling functions and a set of functions from the BLAS API. The frontend is flexible with respect to the number, and composition, of backends, where it is up to the user to create and add backends. Once the frontend has one or more backends, it can start scheduling tasks.

Our data-handling functions transfer data between the frontend and the different backends. They create and maintain matrix objects identified by unique IDs. The IDs correspond to matrices previously created by `new_matrix`:

```
MatrixId new_matrix(int rows, int cols, void* data,
   Precision precision, Type type, StorageOrder order,
   Transpose transpose, Copy copy);
```

The first three arguments to `new_matrix` are the matrix dimensions and a pointer to the matrix data. Data allocation is left to the user. The next four arguments are BLAS specific; the data precision, the matrix type, the data storage order, and whether the matrix is transpose. The storage order is assumed to be column-major as currently required by CUBLAS. The final argument is whether the data should be copied to backend specific storage. Setting that the backend does not need to copy the matrix is used when the matrix is an output argument. This prevents the GPU backend from copying data that will be overwritten to the graphics card. The call to `new_matrix` creates a task, but does not schedule it; the task is only scheduled to a specific backend when it is a dependency of another task. The specific details for scheduling these tasks are outlined towards the end of this section.

As the tasks are executed asynchronously, and the user is unaware what backend performs the actual computation, the function `get_matrix` is supplied:

```
void get_matrix(MatrixId id, bool synchronous);
```

The arguments to `get_matrix` are the matrix ID and a *blocking* flag. The `get_matrix` function creates a task to retrieve the matrix from the backend, and the task is immediately scheduled to the backend that holds ownership of the matrix. If the blocking flag is set, the frontend waits until the task has been executed. The CPU backend simply ignores these tasks, as all its data already resides in CPU memory space. The GPU, however, must read the data back from the GPU to CPU memory if the GPU data has been altered. Using `get_matrix` without the blocking flag enables asynchronous read-back of data.

The last data-handling function offered by the frontend is `delete_matrix`, which creates a task that deallocates backend specific resources occupied by the matrix:

```
void delete_matrix(MatrixId id);
```

For the CPU backend this function does nothing, while the GPU backend frees GPU memory consumed by the matrix. The user is responsible for deallocation of the data pointer supplied to `new_matrix`.

The three functions described in the previous paragraphs enable the user to create, read and delete matrices with our asynchronous API. We will now describe how computational tasks are created, scheduled and executed using the Level 3 BLAS double precision general matrix multiply (dgemm) algorithm as an example. The CBLAS [14] function `dgemm` is defined as

```
void cblas_dgemm(const enum CBLAS_ORDER Order,
        const enum CBLAS_TRANSPOSE TransA,
        const enum CBLAS_TRANSPOSE TransB,
        const int M, const int N, const int K,
        const double alpha, const double *A, const int lda,
        const double *B, const int ldb,
        const double beta, double *C, const int ldc);
```

which computes

$$C \leftarrow \alpha \mathrm{op}(A)\mathrm{op}(B) + \beta C, \quad \mathrm{op}(X) = \{X, X^T\},$$

using the three matrices A, B and C. Our frontend signature is slightly different,

```
void dgemm(int m, int n, int k,
        double alpha, MatrixId aId,
        MatrixId bId,
        double beta, MatrixId cId);
```

but performs the same computation. We have removed the `Order`, `TransA` and `TransB` arguments, as these are incorporated into the `new_matrix` function. We have also altered the way input and output matrices are sent to the function. Instead of sending raw data pointers, we send matrix IDs that correspond to matrix objects.

The implementation of `dgemm` in our frontend is quite simple. We start by looking up the three matrices identified by the matrix IDs. Then we create a `dgemm` task to hold the parameters and invoke `schedule` to schedule the task to a backend.

The `schedule` function has two tasks: first to group the incoming task with its dependencies into a cluster, and second to find the optimal backend to schedule this cluster to. The grouping is trivial, as all tasks have a dependency list. Selecting the optimal backend to schedule the cluster to, however, is nontrivial. We use two criteria to give an estimate of how good a candidate each backend is, and then schedule the cluster to the backend with the best score. The first criterion is based on the load for each backend, and the second on where dependent tasks have been scheduled.

Our load criterion tries to equalize the load of different backends. It consists of three parts: wall time spent computing, an estimate of wall time needed to process the current task queue, and an estimate of wall time needed to compute the incoming cluster. The time needed to process future tasks is estimated by multiplying the average time to compute a single task by the number of tasks in queue. This is a good approximation if the tasks require similar processing time, such as many matrix multiplications of equally sized matrices. There is one problem, however. When we start our frontend, we do not have enough data to calculate a valid average time needed to compute a single task. For a heterogeneous processing environment, the different backends can have orders of magnitude of difference in performance. This is where our automatic tuning comes into play. We have a pre-compilation step that benchmarks each backend to find a static average. We calculate this average by computing several matrix multiplications for a fixed matrix size for each backend. This gives a relationship between the processing power of the different backends. We incorporate this speed estimate into our source code by defining a C preprocessor macro, before recompiling the program. This constant estimate is used when the number of tasks processed by each backend is too low to give a proper run-time estimate.

The second criterion is computed by estimating the time it takes to move existing dependencies from another backend to the current backend. As you typically perform many operations on the same data, you will encounter situations where dependencies reside on different backends. Moving data between CPU backends is relatively inexpensive. Moving data between a CPU and a GPU, however, is very expensive, and should be penalized by this criterion. For each backend, we loop through the dependencies and estimate the time needed to move each dependency. If the dependency already resides on the current backend, the cost is zero. For dependencies on other backends, however, the cost is calculated as the estimated time needed to get the matrix from its current backend and transfer it to the new backend. The time needed to get the matrix requires a synchronous get, and is accordingly very expensive.

When the best backend has been selected based upon the above mentioned criteria, we synchronously get the dependencies that have to be moved. Then we reschedule them to the selected backend together with unscheduled dependencies, followed by the computational task itself. Continuing our `dgemm` example, this means that the `new_matrix` tasks for the matrices `a`, `b`, and `c` will be enqueued to the selected backend just before we enqueue the `dgemm` task.

4.2. Back-end. Each backend has two parts, as indicated by Figure 4.1. It consists of two threads of execution: the frontend thread and the backend thread. The frontend thread is shared by multiple backends, while each backend runs in a distinct thread of execution. Functionality for enqueuing tasks and querying load status is only called from the frontend thread, and functionality for dequeuing and executing tasks is only called from the backend thread. One can view this relationship between the frontend thread and the different backend threads as a typical *boss-worker* relationship. The frontend acts as the boss, distributing tasks among the workers. In this section we use the term *boss* to signify the thread of execution that runs the frontend, and *worker* to signify the thread of execution that runs the backend selected by the scheduler for the current cluster.

When a task is scheduled to a specific worker, the boss simply adds the task to the task-queue of the selected worker. The boss then notifies the worker of a change, triggering the worker to check its status. While the queue is empty, the worker simply idles waiting for this notification. When notified, and a task has been added to the queue, the worker starts dequeuing and processing the tasks. For the dequeued task, it starts by identifying the type (`new_matrix`, `delete_matrix`, `dgemm`, etc.). When the task has been identified, the worker executes the corresponding function, and continues to process the rest of the queue.

In Section 4.1 we explained how the scheduler enqueued a cluster of tasks to a backend based on different criteria. For the dgemm example, the cluster contained the tasks to create the three matrices, `a`, `b`, and `c`, and the `dgemm` task that used these matrices. We continue the example by explaining how the backend processes and executes these tasks in its queue. Assuming that all four tasks have been added to the task-queue of the GPU backend, we start by dequeuing the first. It is the task to create matrix `a`. The GPU backend allocates GPU memory, and transfers the matrix data into the newly allocated memory. We follow the same procedure for matrix `b`, but for matrix `c` we only allocate memory, assuming we have set the copy flag to false when creating the task. We continue by dequeuing the dgemm task. This instructs us to call the CUBLAS dgemm function using the parameters and matrices defined by the task. When we have completed the dgemm task, we check the queue, and, if empty, wait for notification from the frontend.

4.3. MATLAB wrapper. We have implemented a MATLAB wrapper to offer a high-level interface to the asynchronous linear algebra API. The wrapper consists of two parts, one written in C++, and one written in the MATLAB language (M). The C++ part is very lightweight and thin, simply translating a MATLAB call into a call to one of the frontend functions. The M part defines a new class in MATLAB, and uses operator overloading for this class. Internally, this MATLAB class calls the C++ part.

MATLAB can execute user-defined MATLAB executable (MEX) files that are programmed using FORTRAN or C/C++. These MEX files can be called from MATLAB as if they were regular MATLAB functions. When a MEX file is called MATLAB executes the mexFunction, the entry point all MEX files must define:

```
void mexFunction(int nlhs, mxArray* plhs[],
                 int nrhs, const mxArray* prhs[]);
```

The arguments to the mexFunction are the number of left-hand arguments, pointers to the left-hand arguments, number of right-hand arguments, and pointers to the right-hand arguments, respectively. A general MATLAB call can be written as

```
[a, b, ..., y, z] = fun(α, β, ..., ψ, ω).
```

If `fun` is the name of a MEX file, `plhs` would contain the arguments `a` through `z` and `prhs` would contain arguments α through ω. In our wrapper, we use a single MEX file, where the mexFunction calls the correct frontend function based on the first argument. Our MATLAB function calls are on the format

```
matrix_id = async(funk_id, ...),
```

where `async` is the name of our MEX file, `matrix_id` corresponds to the internal ID used by our asynchronous API, and `func_id` determines what function the wrapper should call. The rest of the arguments are sent to the frontend function is specified by `func_id`.

When creating matrices, the C++ part of the wrapper automatically makes the memory allocated by MATLAB *persistent.* This prevents MATLAB for using automatic garbage collection, effectively giving us ownership of the data.

MATLAB has support for classes in its M script language. This enables us to call the MEX file in a more elegant way than explicitly calling `async` as indicated above. The M part of our wrapper consists of a new class, called *Matrix*, that uses operator overloading to offer an API with familiar MATLAB syntax.

Our operator overloading for the Matrix class translates operations and function calls involving a Matrix object into calls to `async`. Matrix multiplication, for example, would be defined in the file `@Matrix/mtimes.m` in the following way:

```
function c = mtimes(a, b)
c.id = async(42, a.id, b.id);
```

Here, 42 is the function id corresponding to dgemm, `a.id` is the id of matrix `a`, and likewise for `b.id`.

The following MATLAB code shows how this is used in practice to compute dgemm:

```
m = 10; n=20; k=30;
a = Matrix(rand(m, k));
b = Matrix(rand(k, n));
c = a*b;
c_data = double(c);
```

First, `a` and `b` are created as two Matrix objects that contain randomly generated data. Then the multiplication is run using operation overloading. Notice that we do not create the matrix `c`. The C++ part of the wrapper handles this by noticing that `c` has an invalid matrix ID, and thus creates a correctly shaped matrix for us. Finally, we convert `c` from a Matrix object back to double-precision data using the overloaded `double` function. This conversion actually performs a synchronous `get_matrix` that will inhibit performance. As our API is asynchronous we also offer a `read` function that corresponds to an asynchronous `get_matrix`. If there is enough time to read back the matrix before the double-precision data is required, we experience very fast data access.

5. Benchmarking and Analysis. The performance of our interface depends on two main factors; the efficiency of the underlying BLAS implementation and the time spent scheduling a task. The efficiency of several BLAS implementations has previously been analyzed and is not discussed here (see e.g., Barrachina et. al. [5] and the NCSA BLAS Performance Comparison). We show how our API behaves with different backend setups, and try to analyze the results. We further benchmark our scheduling algorithm, and the MATLAB wrapper of the API.

We utilize CUBLAS in our GPU backend. CUBLAS is part of the NVIDIA CUDA SDK, and implements single and double-precision BLAS functions using the GPU. It is steadily being developed, and provides a good interface to GPU accelerated BLAS functions. For our CPU back-ends, we use a single-threaded ATLAS implementation.

We have benchmarked our algorithms on two setups. The first setup consists of a 2.4GHz Intel Core 2 Q6600 quad-core processor with eight GiB of 12.8 GiB/s system memory, and an early engineering sample of the next generation NVIDIA Tesla T10 card (not full performance). The second system consists of a 2.4GHz Intel Core 2 E6600 dual-core processor with four GiB of 12.8 GiB/s system RAM, and an NVIDIA GeForce 8800 GTX graphics card (G80).

Our benchmark runs one thousand computations using 1000×1000 matrices, and timing results have been consistent throughout our benchmarking. The following is a pseudo code of the benchmark:

```
start timer;

for 1..10
  create input and output matrices;
  schedule computation on matrices;

for 11..990
  wait for result;
  delete matrices;
  create input and output matrices;
  schedule computation on matrices;
```

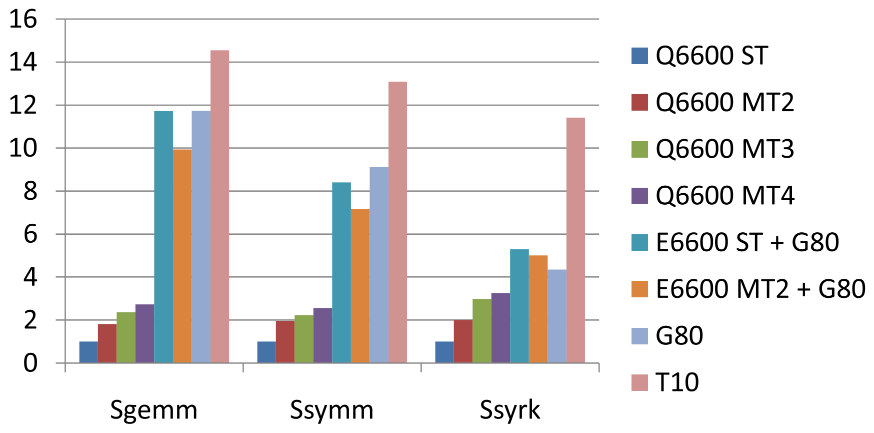


FIG. 5.1. *Speedup results from benchmarking single-precision general matrix multiplication (sgemm), symmetric matrix multiplication (ssymm), and rank-k symmetric matrix update (ssyrk). Please note that the T10 card is an early engineering sample (not full performance).*

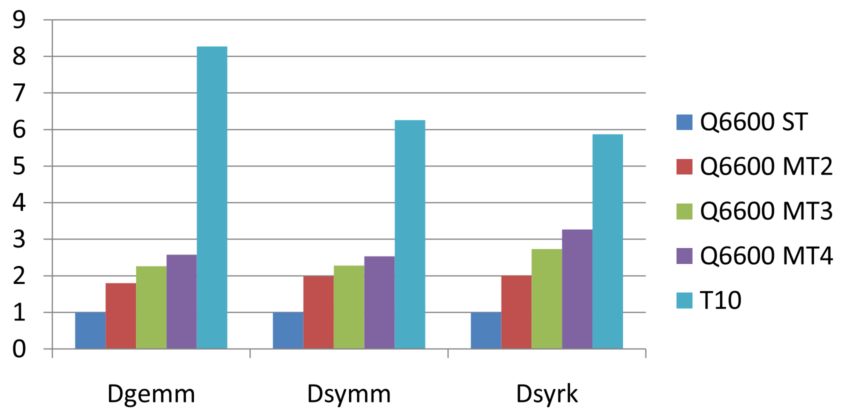


FIG. 5.2. *Speedup results from benchmarking double-precision general matrix multiplication (sgemm), symmetric matrix multiplication (ssymm), and rank-k symmetric matrix update (ssyrk). Please note that the T10 card is an early engineering sample (not full performance).*

```
for 991..1000
  wait for result;
  delete matrices;

stop timer;
```

This is the worst case scenario, where we only run one computation per matrix. This severely affects the GPUs performance, as transferring data between CPU and GPU memory is expensive. In real world use, one would typically reuse data in multiple computations.

We have implemented and benchmarked three algorithms from BLAS in both single- and double-precision: general matrix multiply, symmetric matrix multiply, and rank-k symmetric matrix update. We have split the results into a single and double-precision part, as the G80 GPU does not support double-precision. Figure 5.1 shows the speedup of different backend setups compared to using a single CPU backend for single precision computations. Utilizing two CPU cores scales almost perfect for all algorithms. The average speedup is 1.92×. For three cores, however, we see a distinct difference between the algorithms. Ssyrk continues to scale almost perfect, with a speedup of 2.98, while sgemm and ssymm achieve a much smaller speedup. We believe this is

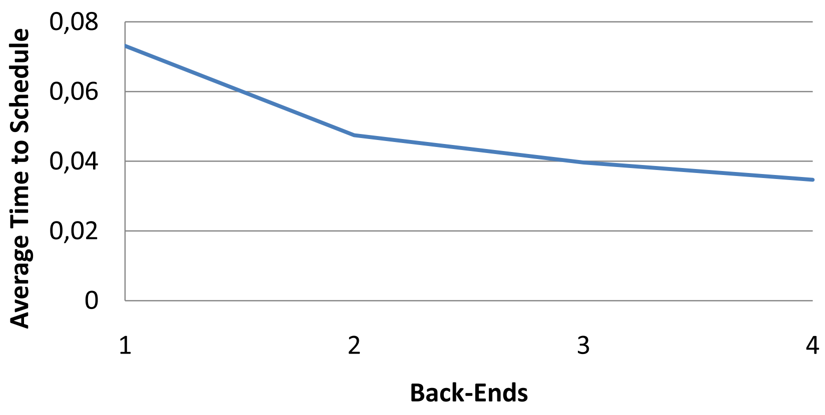


FIG. 5.3. *Average time to schedule a single task compared to the number of backends*

because these two algorithms start trashing the cache. The Q6600 processor has a shared L2 cache of 4MiB per core pair. For two CPU backends, each backend can run on a separate core-pair with exclusive access to the 4MiB cache. For three backends, however, two of the backends must run on the same core-pair, having to share the cache. Ssyrk only uses two thirds of the memory sgemm and ssymm uses, and does not seem to be affected by having to share the processor cache.

When we have four backends running simultaneously, we do not have enough processing power for the scheduler thread to keep all back-ends occupied. Therefore, for four backends, none of the backends scale perfectly, and we do not get the expected speedup.

Figure 5.2 displays the gained speedups for double precision computations, showing similar results to our single-precision equivalents.

Utilizing the GPU backend outperforms using multiple CPU backends by far. This even though we transfer data to and from the GPU for each pass. Utilizing a GPU backend and CPU backend, however, does not scale as well as we could have hoped. We actually see a speed-down compared to using just the GPU for sgemm and ssymm. The reason for this is our scheduling algorithm. By examining the choices made by our scheduler, we find that it makes sub-optimal decisions when faced with a heterogeneous set of backends. We have currently not resolved this issue, but feel confident that we will remove it.

Figure 5.3 shows the average time spent scheduling a task compared to the number of backends. Between 0.04 and 0.08 seconds is a relatively long time. For small matrices this constant scheduling time will affect the experienced performance. Nevertheless, when the tasks we schedule take on the order of seconds to complete, this overhead is negligible. Our API is designed for computation on large matrices, making this overhead less important. However, we believe that optimizing the scheduling algorithm and front- and backend interaction will decrease this overhead.

Our MATLAB wrapper tries to be light-weight and add as little overhead as possible. Our benchmarks show that using our MATLAB wrapper only imposes a constant overhead. Calling our MEX file directly (without the M part of the wrapper) imposes an overhead of 0.3 milliseconds. Using the overloaded functions in the M wrapper is more expensive, as MATLAB must look up the correct function before the calls the C++ part of the wrapper are made. Nevertheless this only imposes an overhead of 2 milliseconds.

There has been skepticism in the scientific community towards using GPUs. It was only recently that GPUs started supporting single-precision arithmetics. These arithmetics, however, do not fully comply with the IEEE standard. Some minor parts such as denormals deviate from the standard, but the floating point implementation has been sufficient in practice for most uses. However, the lack of double-precision numbers has been criticized by the scientific community. Some algorithms require double, or even higher, precision. One example is numerical linear algebra, where even small floating point rounding errors can give large deviations in the results. Using double-precision improves the stability in this respect.

The NVIDIA Tesla T10 GPU supports double-precision in hardware. We do not give a thorough analysis of the double-precision implementation, but offer our early experiences with it. Figure 5 shows the absolute

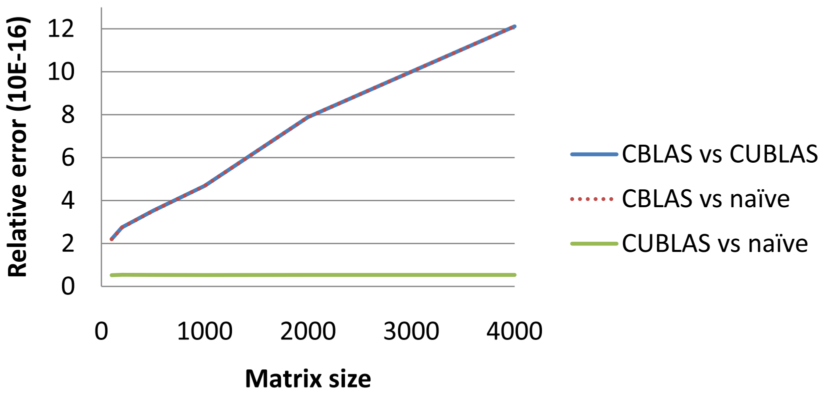


Fig. 5.4. *Measured relative difference per matrix element between CBLAS, CUBLAS and a naïve implementation of double-precision matrix multiplication.*

difference between CBLAS, CUBLAS, and a naïve (double for-loop) CPU implementation of double-precision general matrix multiply (dgemm). There seems to be a linear correlation between the matrix size and the difference between CUBLAS and CBLAS results. To put this into perspective, we also show the difference between CUBLAS and our naïve dgemm implementation. The difference per element between CUBLAS and our dgemm is constant at around $5 \cdot 10^{-17}$. This suggests that the CBLAS implementation reorders operations (e.g., using *Strassens algorithm*) giving different rounding errors than CUBLAS and the naïve algorithm. Reordering operations on the GPU in the same manner (if possible) should thus lessen the experienced difference between CUBLAS and CBLAS.

We will experience that results change between different runs using our API. This is because we do not know a priori where the results will be computed, and the fact that there is a small difference between CBLAS and CUBLAS results. For many uses, this is irrelevant, but for regression testing, for example, one must be aware of the possibility of changes in the result.

6. Conclusions and Future Work. We have presented a task-parallel asynchronous API for numerical linear algebra that imposes only small overheads. This API scales almost perfect to two CPU cores, and is capable of utilizing GPU processing resources for even higher performance. We have also demonstrated that the API can be used through high-level languages, such as MATLAB.

Our scheduling algorithm shows a proof-of-concept that works well for a homogeneous processing environment, but it is suboptimal for a heterogeneous composition of backends. It will be an interesting task to alter the scheduling algorithm to better utilize such heterogeneous processing platforms.

7. Acknowledgements. We would like to thank NVIDIA for the early engineering sample of the next generation Tesla T10 GPU.

REFERENCES

[1] AMD Corporation, *Brook+ language specification, version 1.0 beta*, May 2008.
[2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK Users' Guide (third edition)*, Society for Industrial and Applied Mathematics, 1999.
[3] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, *The landscape of parallel computing research: A view from Berkeley*, tech. report, EECS Department, University of California, Berkeley, 2006.
[4] M. Baboulin, J. Dongarra, and S. Tomov, *Some issues in dense linear algebra for multicore and special purpose architectures.*, tech. report, 2008.
[5] S. Barrachina, M. Castillo, and E. S. Q.-O. Francisco D. Igua and, R. Mayo, *Evaluation and tuning of the level 3 CUBLAS for graphics processors*, in Workshop on Parallel and Distributed Scientific and Engineering Computing, 2008.
[6] S. Barrachina, M. Castillo, F. D. Igual, R. Mayo, and E. S. Quintana-Ortí, *GLAME@lab: An M-script API for linear algebra operations on graphics processors*, 2008.

[7] P. Bientinesi and E. S. Q.-O. et al., *FLAME@lab: A farewell to indices*, tech. report, The University of Texas at Austin, Department of Computer Sciences, 2003. Draft.

[8] A. R. Brodtkorb, *The graphics processor as a mathematical coprocessor in MATLAB*, in CISIS 2008 The Second International Conference on Complex, Intelligent and Software Intensive Systems, 2008.

[9] A. R. Brodtkorb and T. R. Hagen, *A comparison of three commodity-level parallel architectures: Multi-core CPU, the Cell BE and the GPU*. Submitted to PARA'08 9th International Workshop on State-of-the-Art in Scientific and Parallel Computing, May 2008.

[10] S. Brown and J. Rose, *Architecture of FPGAs and CPLDs: A tutorial*, IEEE Design and Test of Computers, 13 (1996), pp. 42–57.

[11] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, *Brook for GPUs: stream computing on graphics hardware*, in SIGGRAPH '04, ACM Press, 2004, pp. 777–786.

[12] D. R. Butenhof, *Programming with POSIX Threads*, Addison-Wesley, 1997.

[13] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, *A class of parallel tiled linear algebra algorithms for multicore architectures (lapack working note 191)*, tech. report, Innovative Computing Laboratory, 2008. `http://icl.cs.utk.edu/plasma/`

[14] J. Dongarra, *Basic Linear Algebra Subprograms Technical forum standard*, International Journal of High Performance Applications and Supercomputing, 16 (2002), pp. 1–111.

[15] J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn, *FLAME: Formal Linear Algebra Methods Environment*, ACM Transactions on Mathematical Software, 27 (2001), pp. 422–455.

[16] IBM, Sony, and Toshiba, *Cell Broadband Engine programming handbook version 1.1*, April 2007.

[17] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, *NVIDIA Tesla: A unified graphics and computing architecture*, IEEE Micro, 28 (2008), pp. 39–55.

[18] W.-K. J. Massimiliano Fatica, *Accelerating MATLAB with CUDA*, 2007.

[19] M. D. McCool, *Data-parallel programming on the Cell BE and the GPU using the RapidMind development platform*, November 2006. GSPx Multicore Applications Conference.

[20] Microsoft corporation, *MSDN DirectX developer center*.

[21] NVIDIA corporation, *CUDA CUBLAS library version 1.1*, September 2007.

[22] ———, *CUDA CUFFT library version 1.1*, October 2007.

[23] ———, *NVIDIA CUDA programming guide version 1.1*, November 2007.

[24] OpenMP Architecture Review Board, *OpenMP application program interface version 3.0*, May 2008.

[25] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, *GPU computing*, Proceedings of the IEEE, 96 (2008), pp. 879–899.

[26] RapidMind, *Cell BE porting and tuning with RapidMind: A case study*. Online. `http://www.rapidmind.net/pdfs/RapidMindCellPorting.pdf`

[27] J. Reinders, *Intel Threading Building Blocks:Outfitting C++ for Multi-core Processor Parallelism*, O'Reilly Media, Inc, 2007.

[28] D. Shreiner, M. Woo, J. Neider, and T. Davis, *OpenGL Programming Guide*, Addison-Wesley, sixth ed., 2007.

[29] J. Stratton, S. Stone, and W. mei Hwu, *MCUDA: An efficient implementation of CUDA kernels on multi-cores*, tech. report, University of Illinois at Urbana-Champaign, 2008.

Edited by: Sabri Pllana, Siegfried Benkner
Received: June 16, 2008
Accepted: July 28, 2008

Scalable Computing: Practice and Experience
Volume 9, Number 3, pp. 165–177. http://www.scpe.org

ISSN 1895-1767
© 2008 SCPE

ON THE POTENTIAL OF NOC VIRTUALIZATION FOR MULTICORE CHIPS[*]

J. FLICH, S. RODRIGO, J. DUATO[†] T. SØDRING[‡] Å. G. SOLHEIM, T. SKEIE, AND O. LYSNE[§]

Abstract. As the end of Moores-law is on the horizon, power becomes a limiting factor to continuous increases in performance gains for single-core processors. Processor engineers have shifted to the multicore paradigm and many-core processors are a reality. Within the context of these multicore chips, three key metrics point themselves out as being of major importance, *performance*, *fault-tolerance* (including yield), and *power consumption*. A solution that optimizes all three of these metrics is challenging. As the number of cores increases the importance of the interconnection network-on-chip (NoC) grows as well, and chip designers should aim to optimize these three key metrics in the NoC context as well.

In this paper we identify and discuss the main properties that a NoC must exhibit in order to enable such optimizations. In particular, we propose the use of virtualization techniques at the NoC level. As a major finding, we identify the implementation of unicast and broadcast routing algorithms to become a key design parameter in order to achieve an effective virtualization of the chip.

The intention behind this paper is for it to serve as a position paper on the topic of virtualization for NoC and the challenges that should be met at the routing layer in order to optimize performance, fault-tolerance and power consumption in multicore chips.

1. Introduction. Multicore chips (processors) are becoming mainstream components when designing high performance processors. As power is increasingly becoming a limiting factor with regards to performance for single-core solutions, designers are shifting to the multicore paradigm where several processor cores are integrated into one chip. Although the number of cores in current processing devices is small (i. e. two to eight cores per chip), the trend is expected to change. As an example, the Polaris chip for TeraFLOP computing from Intel has recently been announced with 80 cores [1]. This chip is a research demonstrator but serves to highlight many of the challenges facing NoC in the multicore era.

As the main processor manufacturers further integrate multicore into their product lines, an increasingly large number of cores is expected to be added to chips. In these chips, a requirement for a high-performance on-chip interconnect (NoC) emerges, allowing efficient communication between cores and from cores to cache blocks and/or memory controllers. Current chip implementations use simple network topologies such as buses or rings [2]. However, as the number of cores increases such networks effectively become bottlenecks within the system, and become impractical from a scalability point of view. For chips with a larger number of cores the 2D mesh topology is usually preferred due to its layout on a planar surface in the chip. This is also the case of the TeraScale chip. In this paper we restrict our view to 2D mesh and 2D torus topologies.

A lot of research has been undertaken on high performance networks over the recent decades and for some NoCs the mechanisms, techniques, and solutions proposed for off-chip networks can be applied directly. However, on-chip interconnects face physical constraints at the nanometer scale that are not encountered (or are not limiting solutions) in the off-chip domain. One example relates to the use of routing tables at the switches which is common for off-chip networks (e.g. InfiniBand [3]). For some NoCs, however, the memory requirements to implement routing tables in switches may not be acceptable.

From our point of view, the main constraints relating to NoCs (and that are not a primary concern in off-chip networks) are: power consumption, area requirements (floor-space), and ultra-low latencies. We note that these concerns suggest that simple solutions for NoCs should be implemented.

A multicore designer must deal with three key metrics when designing a multicore chip (see Figure 1.1): *performance*, *fault-tolerance*, and *power consumption/area*. Achieving a solution that optimizes these metrics is challenging. If we take fault-tolerance as an example, a solution for fault-tolerance may have a negative impact on performance resulting in increased latency. Such a solution may be acceptable and perhaps even desirable for an off-chip network but for a NoC it may be prohibitive. Another example is that for some NoCs, an efficient implementation of a routing algorithm that delivers high throughput may consume too many resources.

[*]This work was supported by CONSOLIDER-INGENIO 2010 under Grant CSD2006-00046, by CICYT under Grant TIN2006-15516-C04-01, by Junta de Comunidades de Castilla-La Mancha under Grant PCC08-0078, and by the HiPEAC network of excellence.

[†]Parallel Architectures Group, Technical University of Valencia, Spain.

[‡]Networks and Distributed Systems Department, Simula Research Laboratory, and Department of Journalism, Library and Information Science, University College Oslo, Norway

[§]Networks and Distributed Systems Department, Simula Research Laboratory, and Department of Informatics, University of Oslo, Norway

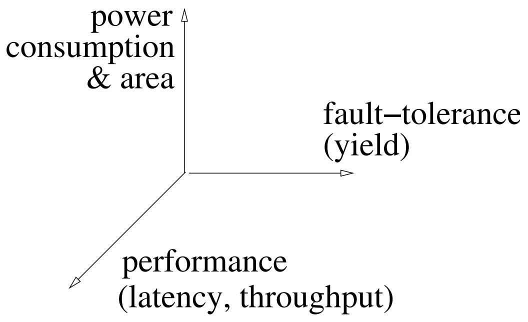


FIG. 1.1. *Design space for a NoC.*

The *Performance* of a network is traditionally measured in terms of packet latencies and network throughput. Currently, within the context of NoC, network throughput is not a major concern as wires can be widened to implement network links where needed. However, ultra-low latencies (less than a few nanoseconds) are typically required and it is usual that a switch forwards a flit within a network cycle. The implication of this is that every stage within the critical path of a packet must be carefully optimized. This is one of the reasons that, in some NoCs, logic-based routing (e.g. the predominant Dimension-Order-Routing, DOR) is the preferable solution as it can lead to reductions in latency as well as power and area requirements.

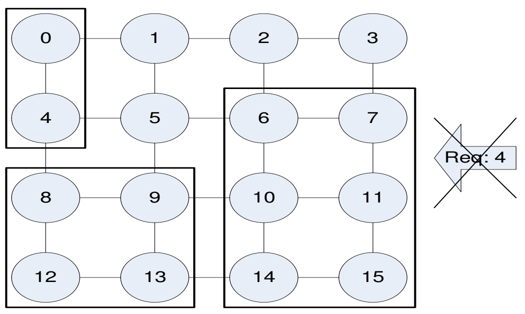
Fault-tolerance is also becoming a major concern when designing a NoC. As the integration of large numbers of components is pushed to smaller scales, a number of communication reliability issues are raised. Crosstalk, power supply noise, electromagnetic and inter-symbol interference are examples of such issues. Moreover, fabrication faults may appear, in the form of defective core nodes, wires or switches. In some cases, even though some regions of the chip are defective, the remaining chip area may be fully functional. From a NoC point of view, the presence of fabrication defects can turn an initial regular topology into an irregular one. In an off-chip network, the defective component(s) can simply be replaced, while for a multicore chip, if the routing layer cannot handle the fault(s), the chip is discarded. This is an issue known as yield and has a strong correlation to manufacturing costs.

Power consumption is also an issue of major concern. As buffers consume both area and energy, a NoC designer must strive to find a balance between acceptable power requirements and the need for buffers. To date, efforts have been made to minimize buffers both in size and number. Another issue related to power is that as the number of cores increases, cores may be under-utilized as applications may not require all the processing logic available on the chip (multimedia extensions for example) and the chip may simply have large idle times for particular applications. An example of the industry response to power consumption is Intel's recent introduction of a new state in their Penryn line of chips that allows the chip to put cores into a deep sleep state where the caches and cores are totally switched off. This points to the fact that mechanisms that dynamically reduce power are both mainstream and will be important in future multicore chips.

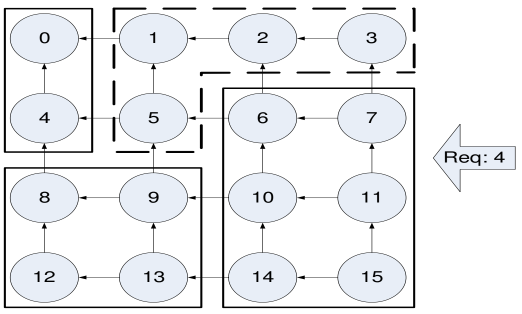
A number of challenging problems are emerging from the increased integration of cores on a chip. When designing a NoC for multicore systems, the three key metrics must be optimized at the same time. In this paper we explore viable solutions to all of them. We propose and argue how the use of virtualization techniques at the NoC level can help deal with the problems discussed above. From a conceptual point of view, a virtualized system provides mechanisms to assign disjoint sets of resources to different tasks or applications. Virtualization is valuable on a multicore chip since it allows for the optimization of the system along the three key design space directions (area/power, performance, yield). For instance, failed or idle components (that can be put to sleep) can be isolated within a proper virtualized configuration, thus providing the potential for higher yields and reductions in power consumption. A virtualized system can separate traffic belonging to different applications or tasks, preventing interference between them, leading to an overall higher performance.

The intention behind this paper is for it to serve as a position paper on the topic of virtualization for NoC and the challenges that should be met at the routing layer in order to maximize performance, fault-tolerance and power consumption.

The remainder of the paper is organized as follows. First, in Section 2 we introduce the concept of virtualizing a NoC and describe the advantages of such an approach in terms of performance, fault-tolerance and



(a) Resource allocations must be sub-mesh shaped, and a request for 4 resource entities are rejected even if 4 resource entities are free (external fragmentation).



(b) Resource allocations must constitute valid Up*/Down* sub-graphs, hence the request for 4 resource entities can be granted (broken line).

FIG. 2.1. *Three tasks are running when another request for resources arrives in a system consisting of 16 resource entities connected in a 4×4 mesh topology: The first task runs on entities number 0 and 4; the second task runs on entities number 6, 7, 10, 11, 14, and 15; and the third task runs on entities 8, 9, 12, and 13.*

power consumption. Then, in Section 3 we provide rough estimations on the impact of a virtualized NoC where we show how a virtualized NoC compares to a non-virtualized one. We conclude in Section 4 with a summary of our findings and discuss future work.

2. Virtualized NoC.

The concept of virtualization is not a new one. It has been applied to different domains for different purposes. A few examples are virtual machines, virtual memory, storage virtualization, and virtual servers in data centers. A virtualized NoC may be viewed as a network that partitions itself into different regions, with each region serving different applications and traffic flows.

Many questions with regard to the design of a virtualized NoC can be posed. Does the virtualized system allow for the merging of different regions? Does the virtualized system support non-regular regions? How is traffic routed within regions? We believe that, in order to meet the design challenges with respect to performance, fault-tolerance, and power consumption, the most important issues for a virtualized NoC solution are: increasing resource utilization; the use of coherency domains; increasing the yield of chips; and reducing power consumption. In this section we provide an overview of each of these issues.

2.1. Increasing Resource Utilization.

Current applications may not provide enough parallelism to feed all the cores that are available on a multicore chip. In such situations some of the cores may be under-utilized for periods of time, with the implication that resources are being wasted. In this situation, different applications (or tasks) should be allowed to use separate sets of cores concurrently. Hence, we need a partitioning mechanism to manage the resources and assign them to the applications in an efficient way.

A partitioning mechanism should, at least, meet the following two challenges, the minimization of fragmentation (maximization of system utilization) and the prevention of interference between messages (packets) that belong to different tasks. The partitioning of a multicore chip can be compared to the traditional processor allocation techniques, and some of the algorithms suggested for processor allocation could also be used for partitioning within a multicore chip. Contiguous allocation strategies (such as [6, 7, 8, 9, 10, 11, 12, 13]) designate a set of adjacent resources for a task. For meshes and tori most of the traditional contiguous strategies allocate strict sub-meshes or sub-cubes and cannot prevent the occurrence of high levels of fragmentation. External fragmentation is an inherent issue for contiguous strategies and occurs when a sufficient number of resource entities are available, but the allocation attempt nevertheless fails due to some restrictions (such as the requirement that a region of available resource entities must constitute a sub-mesh).

Some contiguous resource allocation algorithms have an attractive quality that we refer to as *routing-containment:* the set of resources assigned to a task is selected in accordance with the underlying routing function, such that no links are shared between messages that belong to different tasks. Routing-containment in resource allocation is important for a series of reasons. Most importantly, each task should be guaranteed a fraction of the interconnect capacity regardless of the properties of concurrent tasks. Thus, if one task introduces severe congestion within the interconnection network, other tasks should not be affected. In Section 3 we discuss results that show the effects of traffic overlapping. In earlier works routing-containment was an issue that was often only hinted at. Even so, many strategies, like those that allocate sub-meshes in meshes, will be routing-contained whenever the DOR routing algorithm is used.

A close coupling between the resource allocation and routing strategies allows for the development of new approaches to routing-contained virtualization with minimal fragmentation. These approaches involve the assignment of irregularly shaped regions and the use of a topology agnostic routing algorithm. In general, these approaches can be used for any topology – a particularly attractive property in the face of faults in regular topologies.

One approach which is particular to the Up*/Down* [16] routing algorithm assumes that an Up*/Down* graph has been constructed for an interconnection network that connects a number of resources. For each incoming task the assigned set of resources must form a separate Up*/Down* sub-graph. This ensures high resource utilization (low fragmentation) since allocated regions may be of arbitrary shapes. In this case routing-containment is ensured without the need for reconfiguration. UDFlex [18] (for which performance plots that demonstrate the advantages of reducing fragmentation are included in Section 3) is an example of such a strategy.

Figure 2.1(a) shows a situation where three processes have been allocated to a NoC that does not support irregular regions. An incoming process that requires the use of four cores has to be rejected as the NoC only supports regions of regular shapes. Figure 2.1(b) on the other hand shows the case where a NoC supports the use of irregular regions. In the first case the DOR routing algorithm is used. Each packet must traverse first the X dimension and then the Y dimension. However, the acceptance of new applications is highly restricted as regions must be rectangular to allow traffic isolation with DOR routing (in Figure 2.1(a) node 5 could not communicate with nodes 2 and 3 without interfering with an already established domain). In Figure 2.1(b), however, the topology-agnostic Up*/Down* routing algorithm is used (node 0 is the root of the Up*/Down* graph). This allows allocation of irregular regions, which increases resource utilization.

In general, topology agnostic routing algorithms are less efficient than topology specific routing algorithms (although, in some cases, they perform almost as well [14, 15, 17]). Thus, when allocating arbitrarily shaped contiguous regions, there is a trade-off between routing-containment and routing efficiency. This is the case both for NoCs and off-chip networks. Also, when allocating arbitrarily shaped regions, reconfiguration may be needed to ensure routing-containment for incoming tasks.

The routing algorithm used in a NoC should be flexible enough to allow for the formation of irregularly shaped regions. The tight constraints applied to multicore chips with respect to issues such as latency, power-consumption, and area limit the choice of strategies available for routing and resource allocation. The use of routing tables requires significant memory resources in switches. Source-based routing, on the other hand, contains the entire path of a packet in its header and allows for a quicker routing decision when compared to the table approach. NoCs that allow routing tables or source-based routing are flexible with respect to the applicability of topology agnostic routing and assignment of arbitrarily shaped regions. For such environments the approaches described above may be used without restrictions, and solutions targeted for off-chip environments may be adopted. For some NoCs, however, the use of both routing tables and source-based routing may be unacceptable or disadvantageous. This group of NoCs can instead use a logic-based routing scheme. For allocation of sub-meshes in a mesh topology, DOR (which can easily be implemented in logic) is a possible choice. More interestingly, under certain conditions, recent solutions such as the region-based routing concept [4] and LBDR mechanism [5] allow the use of topology-agnostic routing algorithms in semi-regular topologies with small memory requirements within switches.

2.2. Coherency Domains. Typically, in a multicore system the L2 cache is distributed among all cores. This is a result of the use of L2 private caches or having a shared L3 on-chip cache (e.g. Opteron with Barcelona core). Each core has a small L2 cache and a coherency protocol is used. As the number of cores within a chip increases several new problems arise. Upon a write or read miss access from a core to its L2 cache the coherency protocol is triggered and a set of actions are taken. One possible action is to invalidate any possible copy of the block in the remaining L2 caches. This is normally achieved by using a snoopy protocol. However, implementing a snoopy protocol in a 2D mesh is difficult as snoopy actions do not directly map on to a 2D mesh.

A possible solution is to implement a snoopy protocol with broadcast packets. Although broadcasting solves the problem of keeping the coherency, as the system increases in number of cores, the broadcast action becomes inefficient. For instance, in a 80-core system broadcasting an invalidation is inefficient if only two or three copies of the block are present in the system. The entire network is flooded with messages that are only directed at two or three cores (and they will probably be physically close to each other). Figure 2.2(a) shows an example

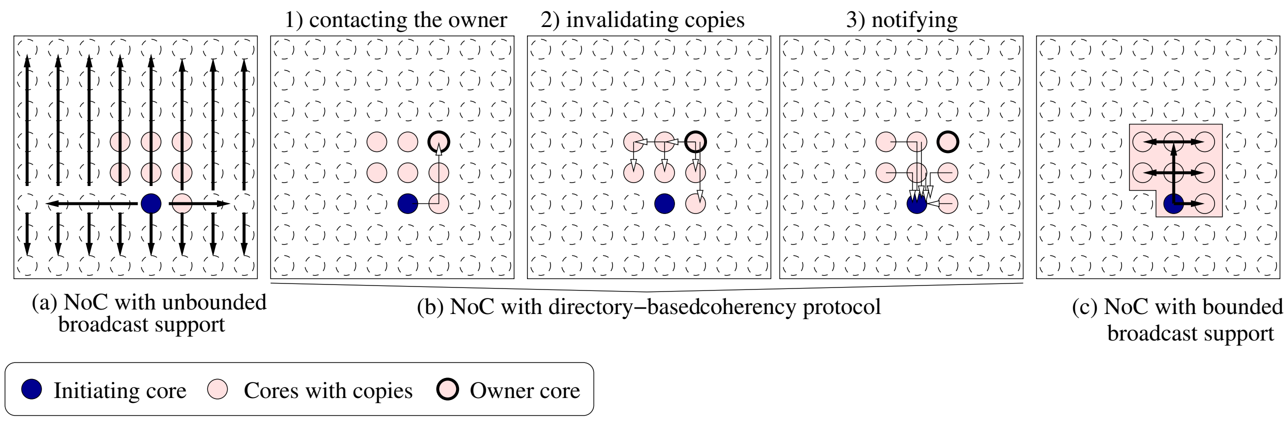

1) contacting the owner
2) invalidating copies
3) notifying
(a) NoC with unbounded broadcast support
(b) NoC with directory−basedcoherency protocol
(c) NoC with bounded broadcast support
Initiating core
Cores with copies
Owner core

FIG. 2.2. *Examples of different coherency protocols.*

of this.

One possible solution is the use of directories to implement a coherency protocol. In this situation, each memory block has an assigned owner. Upon an action on a block the owner is inspected. The owner has a directory structure of all the cores that have copies of its block. Thus, the packet is sent only to the cores with copies of the accessed block. However, directories require significant memory resources that may be prohibitive for chip implementations as there are increases in latency due to indirection (see Figure 2.2(b)).

One solution that is becoming a reality is the use of coherency domains. When an application is started on a multicore system different threads or processes are launched on different cores and a possible solution is to map groups of cores that share data in a unique coherency domain. By implementing the required resources at the network and application level, the coherency protocol could remain enclosed within the coherency boundaries and thus prevent interference with traffic in other parts of the chip. Additionally, snoopy actions should be enclosed within the coherency domain by a restricted broadcast. In Figure 2.2(c) the snoopy action (implemented as a broadcast) is bounded to the coherency domain.

It is clear that within the context of virtualization for NoC, mechanisms that allow the use of localized broadcast actions within a coherency domain (network region) should be implemented. Additionally, it may be possible that some resources within the chip are shared by different domains, an example of this is memory controllers. In this situation the virtualized NoC must provide solutions to bound the traffic of each domain within its limits. This is required also for broadcast packets so each domain does not flood all the shared domains.

2.3. Yield and Connectivity of Chips. The manufacturing process of modern day chips consists of many stages but involves photo-lithographic processing of multiple logical as well as physical layers. Many factors can influence the quality of a chip and as high volume manufacturing technology scales from its current 45 nm to 32 nm and below, the occurrence of tile-based interconnect faults may become a greater issue to yield. Yield is defined as the number of usable chips versus defective chips that come off a production line, and it is expected that, as we go deeper into sub-micron processor manufacture, we will see decreases in yield [20].

A modern CPU consists of multiple metal layers, and it is common to use at least 10 layers to make an interconnect for a single core chip. When discussing yield and interconnects, it is important to make the distinction between the interconnect that makes up a single core within a chip (existing between multiple metal layers) and the global routing interconnect that is used between tiles.

Yield as a connectivity issue can benefit greatly from virtualization. Recall that once a chip has been manufactured, faulty components cannot simply be swapped out. NoCs need to be able to sustain faults that arise during production. Figure 2.3 shows two different fault situations. In the first case an entire block of the chip has been disabled as a result of a manufacturing defect. This chip can be recovered by the definition of a virtualized NoC with cores being grouped in a network region.

In the second case, however, a switch is disabled at the middle of the NoC. Two solutions arise here. First, the failed component can be excluded by a smart partitioning scheme that partitions the NoC into different regions. In a second solution, some simple static fault-tolerance mechanisms can be added to switches to circumvent the failure within a region. We expect that solutions to yield can support one component failure.

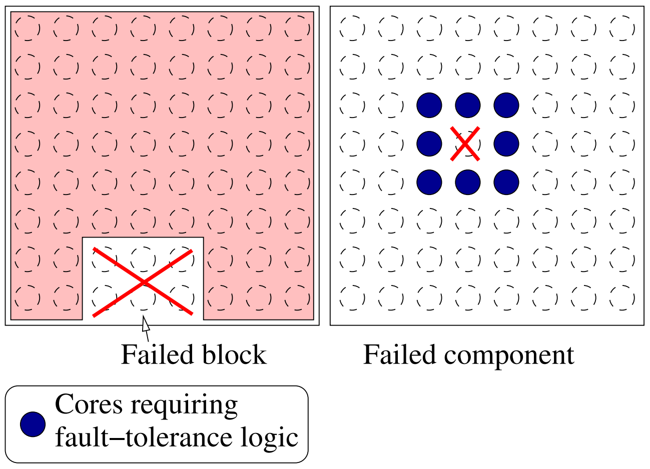


FIG. 2.3. *Dealing with component failures.*

An analysis of yield and connectivity was carried out in [19]. Two straight forward approaches to increasing yield as a result of faults was examined. These methods are shown in Figure 2.4.

The first, **D**imension-**O**rder **R**outing that supports **F**aults (DOR-F) that ensures continued connectivity when a fault is detected, is a mechanism used by the BlueGene/L torus network [22, 21]. This approach requires that the routers within each tile of the disabled area are prohibited from forwarding or accepting traffic that have the tile as destination or source. Rather it can solely act as a transit point forwarding traffic according to the DOR-YX algorithm. Hence the processing nodes are unreachable and can be switched off. As an example, with reference to Figure 2.4(a), tile 12 has been discovered to be faulty so tile 7 only forwards in the horizontal dimension. Similarly tile 13 only forwards in the vertical dimension. By doing so the remaining tiles within the NoC are guaranteed connectivity.

As DOR routing algorithm is still in use, no new turns are introduced to the channel dependency graph (according to the original DOR-YX) and hence the network retains its deadlock freedom. This method is referred to as DOR-F.

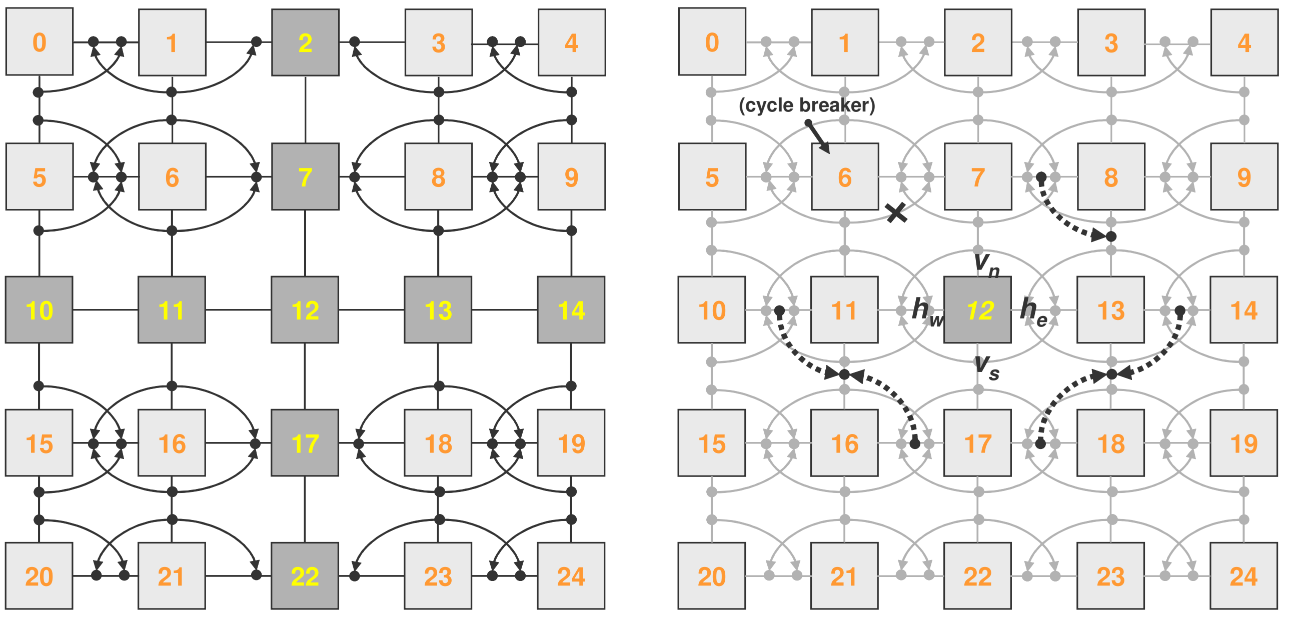
Another approach to the problem is based on an analysis based on **C**hannel **D**ependency **G**raphs that support **F**aults (CDG-F). The CDG-F method allows for DOR-YX routing in the fault-free case and when a packet's route does not cross a fault, but in the case where a packet crosses a fault it uses a re-routing mechanism. Figure 2.4(b) shows an example where tile 12 is faulty, and the five dependencies that must be added and the single dependency that must be removed in order to maintain connectivity and freedom from deadlock.

2.4. Reducing Power Consumption. Power consumption is a major concern. There are potential savings that can be made in effective power-aware routing techniques, that optimize the application traffic to ensure it finishes as quickly as possible and allow the network and cores to shut-down inactive parts.

The literature reports the interconnect power consumption at approximately 30% to 40% of total chip power consumption, a figure that also relates to the TeraScale chip. One reason the interconnect power consumption rate is so high is that the processing cores are not logic intensive (they are not fully fledged x86 cores). If this ratio was to significantly decrease then power aware routing may be less interesting.

Virtualization is also a power issue. In Section 2.1 we discussed issues relating to the sub-optimal allocation of processes to cores. If the operating system has tasks waiting to be allocated and is unable to allocate the tasks to the resources because of fragmentation, then the sub-optimal allocation strategy is not power-efficient and the operating system should ensure (fragmented) processing resources are put to sleep.

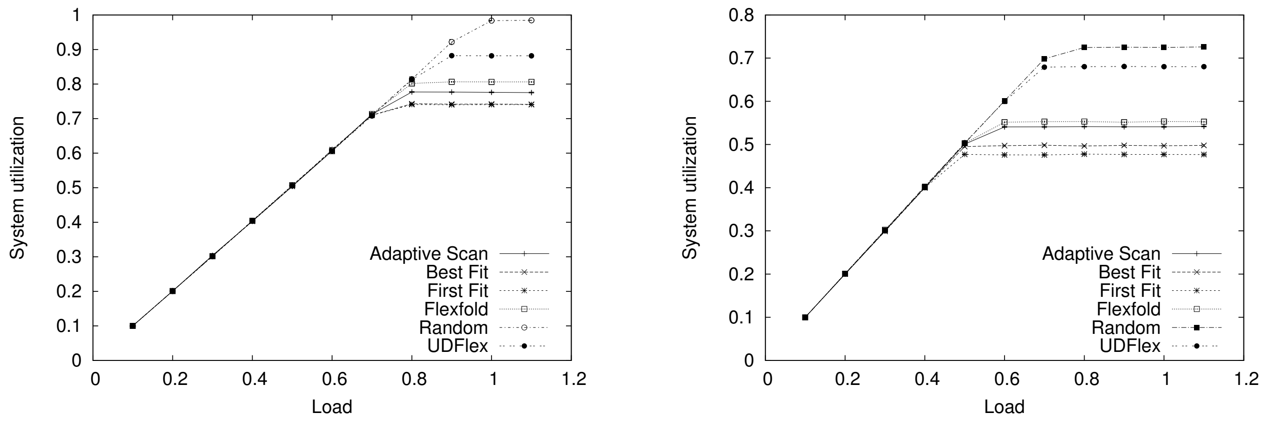
Virtualization also provides solutions to the power issue for NoC in the same way that it supports increases in yield by supporting fault-tolerance. Cores and interconnect components that are otherwise idling can be put to sleep and excluded at the NoC level by using the concept of virtualization, but in this setting, the virtualization technique has to be dynamic in nature, while from the yield point of view it could be static.



(a) The DOR-F. All switches/routers in the same row / column as switch 12 can only transit traffic

(b) The CDG-F method. New dependencies added to route deadlock free around switch 12

FIG. 2.4. *Illustrating two straight forward approaches that can help the yield/connectivity issue. Switch 12 is deemed faulty*



(a) Maximum task size is 16.

(b) Maximum task size is 256.

FIG. 3.1. *Effect on fragmentation* (1 − *system utilization*) *when allocating irregular regions. System utilization for a* 16×16 *mesh.*

3. Basic Examples. This section provides a basic set of examples with performance analysis to motivate virtualization of a NoC. First, we focus on the reduction of fragmentation effects when allowing allocation of irregular regions. Later, we evaluate the effects on performance when using traffic isolation through a virtualized NoC, either using unicast and multicast traffic. Finally, we analyze the yield improvement when using different routing algorithms.

3.1. Reducing Fragmentation Effects. Section 2.1 explained the fragmentation issue related to allocating contiguous regions of resources. Restricting allocations to regions of a particular shape (such as sub-meshes) results in severe fragmentation and, thus, low resource utilization. We argued that allowing irregularly shaped regions may reduce fragmentation, and introduced an approach, called UDFlex, that allows allocation of irregular regions (by allocating Up*/Down* sub-graphs). In Figure 3.1 the system utilization of UDFlex is compared with that of several strategies that allocate sub-meshes (Adaptive Scan, Flexfold, Best Fit and First Fit) for a

16×16 mesh topology. Random is a non-contiguous fragmentation-free strategy, and, thus, a theoretical upper bound performance indicator with respect to system utilization.

Each task requests $a \times b$ resource entities (which is considered a product of numbers for the strategies that allow allocation of irregular regions –UDFlex and Random– and a sub-mesh specification for the other strategies). a and b are drawn from separate uniform distributions with maximum sizes a_{max} and b_{max}, respectively. Figures 3.1(a) and 3.1(b) show system utilization for tasks with $\{a_{max} = 4, b_{max} = 4\}$ and $\{a_{max} = 16, b_{max} = 16\}$, respectively. In both cases the increased flexibility of allocating irregular regions reduces fragmentation and increases the utilization of system resources when compared to sub-mesh allocation. We note that the advantage of allocating irregular regions is even more pronounced for the larger task sizes. A study of the torus topology gave similar results as for mesh.

3.2. Performance Impact From Traffic Isolation: Unicast Packets. Traffic isolation is the main property a virtualized system must achieve. In such a situation, traffic from one domain is not allowed to traverse other domains. To see the performance effects of a non-isolated traffic situation, we compare different routing algorithms and their behavior in a 8×8 mesh split into two domains (see Figure 3.2).

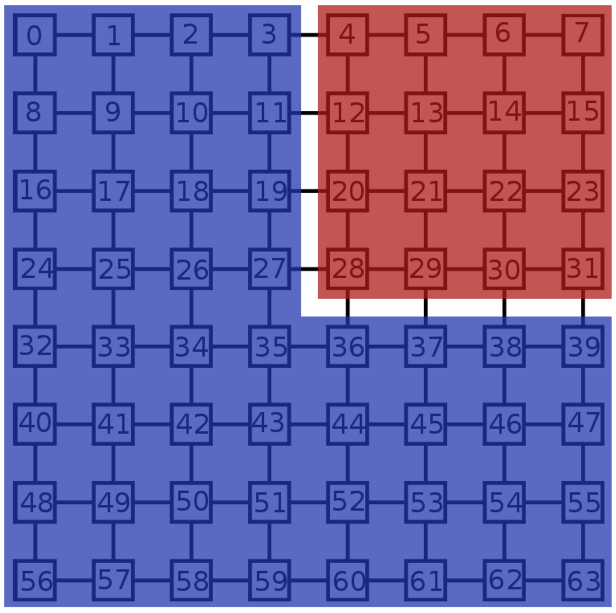
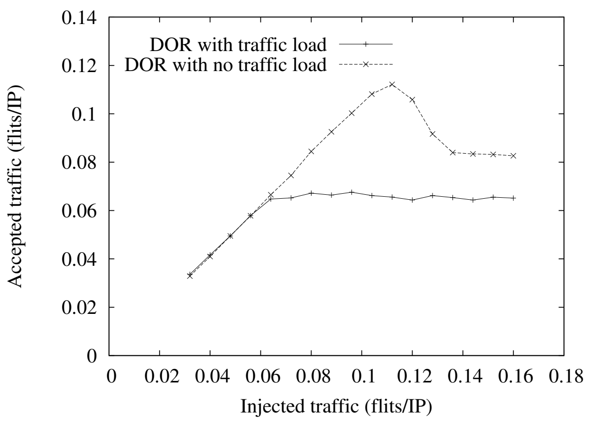


Fig. 3.2. *Domains evaluated for traffic isolation on a* 8×8 *mesh.*

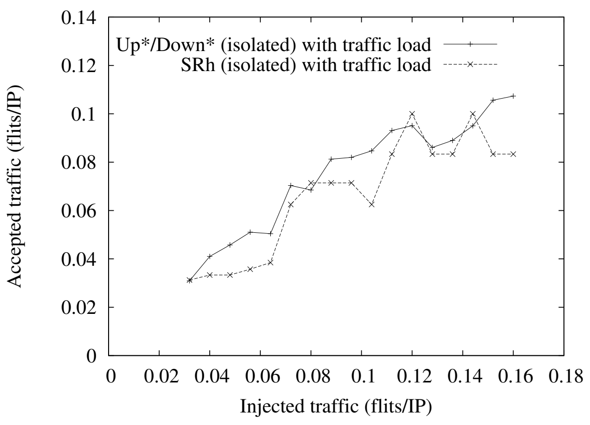
The first domain consists of the top-right 4×4 sub-mesh. This domain is tested with no traffic load (this is plotted as "no load" in the figures) and with a congested situation (this is plotted as "with load" in the figures). The remaining network belongs to a second domain where we inject the full range of traffic from low load to saturation. We have evaluated the following routing algorithms: *DOR*, SR_h and *Up*/Down**. Recall that *DOR* can not sustain routing containment in an irregular topology, so it can not prevent interference between both domains. For SR_h and *Up*/Down**, however, traffic is isolated within the domains. In all the evaluations wormhole switching is assumed.

In Figure 3.3.a we can observe the effects on performance when traffic isolation is not enforced by the routing algorithm. When using *DOR* there are many packets that must traverse the congested domain (top-right 4×4 sub-mesh). As can be seen, network throughput dramatically drops to 0.07 flits/IP. In the absence of traffic in the sub-mesh, network throughput would achieve 0.11 flits/IP, instead. Note that SR_h and *Up*/Down** (see Figure 3.3.b) which enforce traffic containment do achieve such throughput numbers even in the presence of congestion in the sub-mesh. In an isolated system, packets do not cross domains and are thus not affected by the congested traffic.

Delay is also observed in Figure 3.4 to be significantly greater when traffic is routed with *DOR*. When the small domain is congested, average packet latency for the large domain is one order of magnitude higher (when

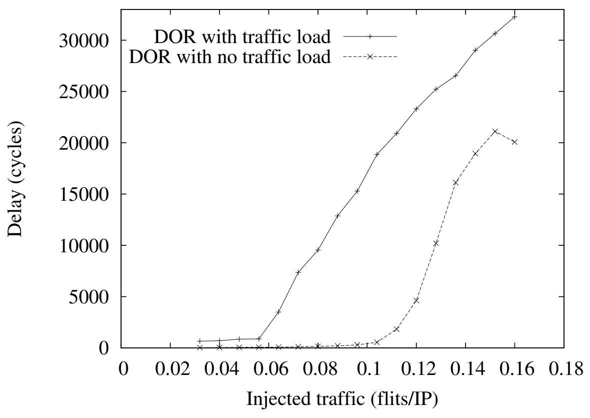


(a) No traffic isolation with *DOR*.

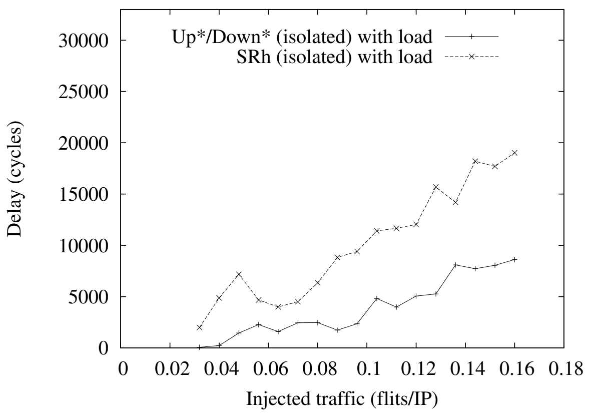


(b) Traffic isolation with SR_h and *Up*/Down**.

FIG. 3.3. *Performance impact from traffic isolation. Network throughput for an* 8×8 *mesh.*



(a) No traffic isolation with *DOR*.



(b) Traffic isolation with SR_h and *Up*/Down**.

FIG. 3.4. *Performance impact from traffic isolation. Network latency for an* 8×8 *mesh.*

compared with the case that the small domain has no traffic). On the other hand, packet latency is much lower for SR_h and *Up*/Down**. With *DOR*, every packet that crosses the congested domain also gets congested, thus extending the congestion to both domains.

3.3. Broadcast and Coherency Domains Analysis. As previously stated, it is desirable for a virtualized system to provide a broadcast mechanism bounded to the domains in which the NoC is partitioned. Unbounded broadcast actions affect the overall network performance and thus may be prone to congestion and undesirable effects (mainly increased packet latency).

For this evaluation, two configurations were used. In the first configuration the entire NoC, a 8×8 mesh, is used with no domain definition, therefore, a broadcast floods the entire network. In the second configuration, however, the mesh is divided into four domains as depicted in Figure 3.5, and a broadcast is initiated within each domain. In the latter case, each broadcast only floods the domain where it was issued. The four broadcasts are issued at the same time from different nodes (9, 30, 36 and 59). The performance has been evaluated both with no background traffic present in the network and with unicast background traffic, randomly distributed within its domain. In each scenario the broadcast latency is measured from the time the broadcast is initiated to the time the last end-node received the message header (i. e. all broadcast actions are finished).

Figure 3.6 shows the results. The first and second columns show the broadcast latency for both the configurations under study when no background traffic is injected. The third and fourth columns show results for both configurations with background traffic. As can be observed, when uniform background traffic is

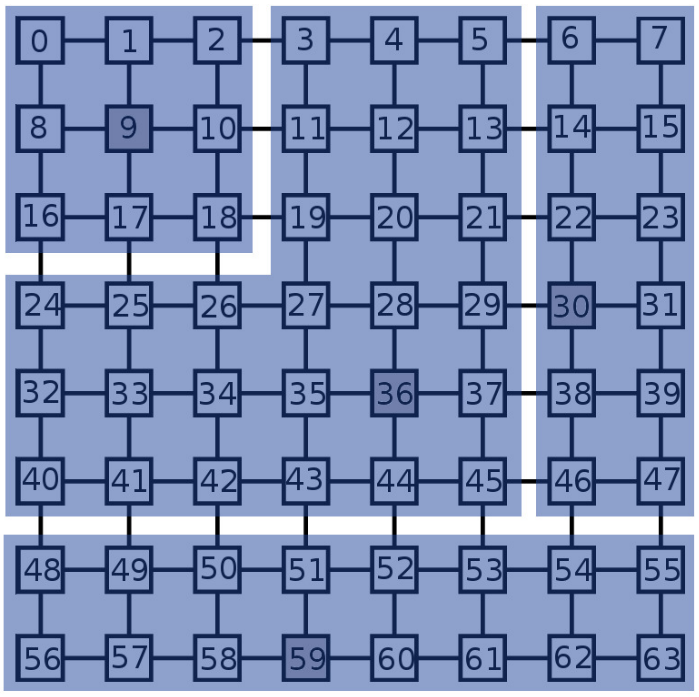


FIG. 3.5. *A* 8 × 8 *mesh divided in 4 domains for bounded broadcast performance evaluation.*

present in the network, using broadcast domains significantly improves the overall performance of the network. In particular, when broadcast domains are used, 239 cycles are required to handle the four broadcasts, whereas 17273 cycles are needed when no broadcast boundaries are used.

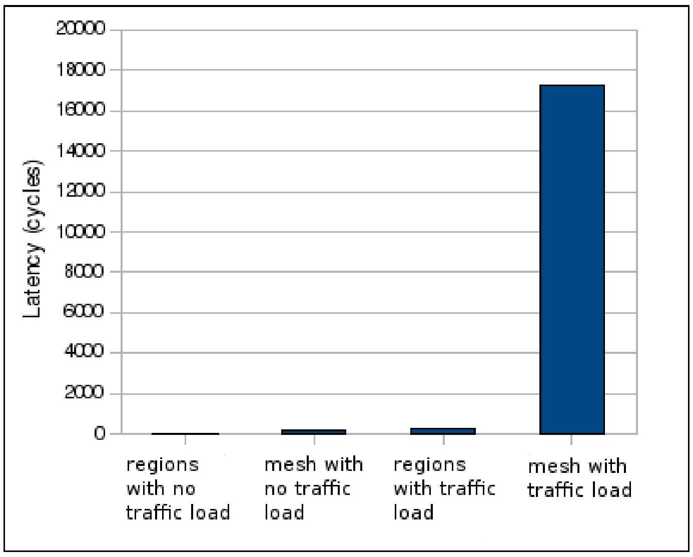


FIG. 3.6. *Effect on latency in a* 8 × 8 *mesh with and without broadcast domains.*

3.4. Yield and Connectivity Analysis. The potential improvements in yield by taking yield into account at the routing level is obtained by increasing connectivity for a NoC with faults and thus the number of usable / routable cores within a NoC. When analyzing yield and connectivity it is important that the chip remains connected and deadlock free. If neither of these conditions are met then we assume that the chip is defective.

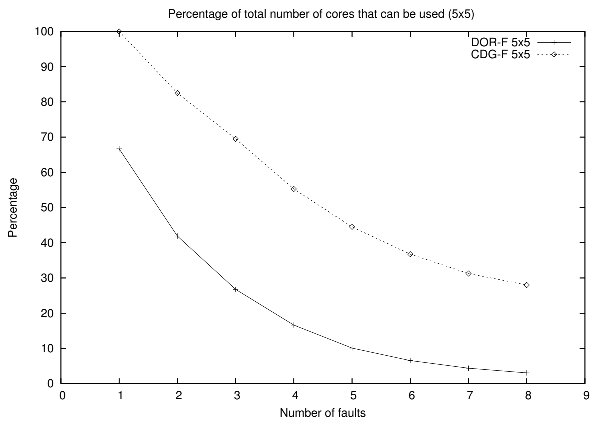
Both the CDG-F and DOR-F methods were tested. For the CDG-F method, an initial channel dependency graph for the given mesh size using DOR-YX routing, without any faults is computed. The desired number of faults is drawn from a random distribution, and for each fault new channel dependencies are added and removed as described in Figure 2.4(b). Dependencies are added to the channel dependency graph as long as it does not introduce a cycle. Once a cycle is found, the chip is assumed to be deadlock-prone and unusable.

Next, a routing-connectivity check is run and involves checking the path from each source to each destination within the chip. The computation of routing paths is based on the channel dependency graph and details whether or not the routing function is connected for a given set of faults. Note that even though there may be physical connectivity on the chip there may not be routing-connectivity. This can be a result of turns that may result in deadlock if such new turns are introduced. We only consider chips where the tiles have physical connectivity.

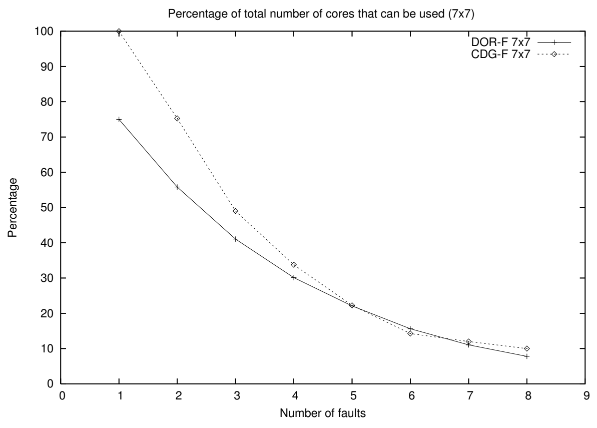
For the DOR-F method the connectivity is analyzed as described in Figure 2.4(a). The DOR-F method will always maintain routing connectivity at an extreme cost associated with switching off processing nodes within tiles.

We analyzed the difference between the two methods (DOR-F and CDG-F). The goal was to draw an over-all picture of the yield situation to see which of the two methods provide the overall highest number of routable (usable) tiles. The answer to this question is shown in Figure 3.7.

Figure 3.7 shows the percentage of tiles that are connected and thus usable / routable for mesh sizes 5×5 and 7×7. Common amongst both plots is the fact that the CDG-F method out-performs the DOR-F for a single fault. With smaller network sizes, CDG-F outperforms DOR-F by a significant margin, and we see the performance cross over point, where both methods exhibit the same performance, in the 7×7 mesh after the occurrence of five faults. For increasing sizes of networks, greater than 7×7, the DOR-F method outperforms the CDG-F method after two faults.



(a) 5×5. CDG-F outperforms DOR-F method



(b) 7×7. Showing the performance cross over point for both methods

FIG. 3.7. *The percentage of connected and thus usable cores plotted against an increasing number of faults.*

These graphs represent the connectivity yield of chips against an increasing number of faults and may be of interest to high volume NoC manufacturers when deciding to adopt a given yield-aware routing strategy.

Manufacturers may already be in a position to anticipate a product-line of certain sizes and ranges but it is unlikely that the exact details of how larger multicore chips will be manufactured has yet been decided.

For situations where only ports within a router have faults, then the interconnect failure can be seen as a link failure, and [23] have detailed how link failures can be handled in mesh networks.

4. Conclusion. Networks on Chip is still a new research area, and many problems are still to be solved. In this paper we have concentrated on three central problem areas, namely power consumption, increasing production yield by utilizing the functional parts of faulty chips, and performance as a measurement of resource utilization. Our position is that these problems can be resolved using virtualization techniques.

The research that is needed to leverage the potential of NoC will be across several fronts: New and flexible routing algorithms that support virtualization need to be developed. We demonstrated how DOR and

Up*/Down* have very different properties with respect to fragmentation under the requirement of routing containment, and we believe that routing algorithms specifically designed for this purpose will be beneficial.

Furthermore, the concept of fault tolerance in on-chip interconnection networks will take on another flavor, as permanent faults on a chip need to be handled with mechanisms that require very little area. Mechanisms that need complex protocols or extra routing tables are therefore of limited interest.

Finally, whereas earlier work on power aware routing has focused on turning off or reducing the bandwidth of links as traffic is reduced, we are now facing a situation where several of the cores on the chip may be turned off. This is a new situation in the sense that where previous methods needed to react to traffic fluctuations over which they had very limited control, we can now develop methods for power-aware routing under the assumption that, to some extent, we can control which cores are to be turned off. This gives an added flexibility in the way we handle the inherent planning of problems with sleep/wake-up cycles.

The intention of this paper is not to provide complete solutions to the problems we address. Rather, we have explained the properties of these problems, quantified some of them through simulations, and explained and demonstrated how virtualization shows promise as a unifying solution to all of them.

REFERENCES

[1] Y. Hoskote, S. Vangal, A. Singh, N. Borkar, and S. Borkar: A 5-GHz Mesh Interconnect for a Teraflops Processor. In *IEEE Micro Magazine*, Sept-Oct. 2007, pp. 51-61.

[2] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy: Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, vol. 49, no. 4/5, 2005.

[3] InfiniBand Trade Association: *InfiniBand Architecture Specification version 1.2.* `http://www.infinibandta.org/specs` 2004.

[4] J. Flich, A. Mejía, P. López, and J. Duato: Region-Based Routing: An Efficient Routing Mechanism to Tackle Unreliable Hardware in Networks on Chip. In *First International Symposium on Networks on Chip (ISNOC)*, 2007.

[5] J. Flich, S. Rodrigo, and J. Duato: An Efficient Implementation of Distributed Routing Algorithms for NoCs In *Second International Symposium on Network-on-Chip (ISNOC)*, 2008.

[6] J. Ding and L. N. Bhuyan: An Adaptive Submesh Allocation Strategy for Two Dimensional Mesh Connected Systems. In *International Conference on Parallel Processing (ICPP'93)*, page 193, 1993.

[7] V. Gupta and A. Jayendran: A Flexible Processor Allocation Strategy for Mesh Connected Parallel Systems. In *International Conference on Parallel Processing (ICPP'96)*, page 166, 1996.

[8] M. Kang, C. Yu, H. Y. Youn, B. Lee, and M. Kim: Isomorphic Strategy for Processor Allocation in k-ary n-cube Systems. *IEEE Transactions on Computers*, 52(5):645–657, May 2003.

[9] F. Wu, C.-C. Hsu, and L.-P. Chou: Processor Allocation in the Mesh Multiprocessors Using the Leapfrog Method. *IEEE Transactions on Parallel and Distributed Systems*, 14(3): 276–289, March 2003.

[10] Y. Zhu: Efficient Processor Allocation Strategies for Mesh-Connected Parallel Computers. *Journal of Parallel and Distributed Computing*, 16(4): 328–337, Dec 1992.

[11] P.-J. Chuang and C.-M. Wu: An Efficient Recognition-Complete Processor Allocation Strategy for k-ary n-cube Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 11(5): 485–490, May 2000.

[12] W. Mao, J. Chen, and W. Watson III: Efficient Subtorus Processor Allocation in a Multi-Dimensional Torus. In *8th International Conference on High-Performance Computing in Asia-Pacific Region*, page 53, 2005.

[13] H. Choo, S.-M. Yoo, and H. Y. Youn: Processor Scheduling and Allocation for 3D Torus Multicomputer systems. *IEEE Transactions on Parallel and Distributed Systems*, 11(5):475–484, May 2000.

[14] O. Lysne, T. Skeie, S.-A. Reinemo, and I. Theiss. Layered Routing in Irregular Networks. *IEEE Transactions on Parallel and Distributed Systems*, 17(1):51–65, January 2006.

[15] J. C. Sancho, A. Robles, J. Flich, P. López, and J. Duato: Effective Methodology for Deadlock-Free Minimal Routing in InfiniBand Networks. In *International Conference on Parallel Processing (ICPP'02)*, pages 409–418, 2002.

[16] M. D. Schroeder, A. D. Birrell, M. Burrows, H. Murray, R. M. Needham, T. L. Rodeheffer, E. H. Satterthwaite, and C. P. Thacker: Autonet: A High-Speed, Self-Configuring Local Area Network Using Point-to-Point Links. SRC Res. Rep. 59, Digital Equipment Corp., 1990.

[17] T. Skeie, O. Lysne, J. Flich, P. López, A. Robles, and J. Duato: LASH-TOR: A Generic Transition-Oriented Routing Algorithm. In *International Conference on Parallel and Distributed Systems (ICPADS'04)*. IEEE, 2004.

[18] Å. G. Solheim, O. Lysne, T. Sødring, T. Skeie, and J. A. Libak: Routing-Contained Virtualization Based on Up*/Down* Forwarding. In *International Conference on High Performance Computing (HiPC'07)*, pages 500–513, 2007.

[19] T. Sødring, Å. G. Solheim, T. Skeie, S.-A Reinemo: An Analysis of Connectivity and Yield for 2D Mesh Based NoC with Interconnect Router Failures. To appear in *11th EUROMICRO Conference on Digital System Design (DSD'08)*, 2008.

[20] N. Campregher, P. Y. K. Cheung, G. A. Constantinides, and M. Vasilko: Analysis of Yield Loss due to Random Photolithographic Defects in the Interconnect Structure of FPGAs. In *International Symposium on Field-Programmable Gate Arrays (FPGA'05)*, pages 138–148, 2005.

[21] M. E. Gómez, J. Duato, J. Flich, P. López, A. Robles, N. A. Nordbotten, O. Lysne, and T. Skeie: An Efficient Fault-Tolerant Routing Methodology for Meshes and Tori. In *IEEE Computer Architecture Letters*, vol. 3, no. 1, 2004.

[22] W. Barrett and The BlueGene/L Team: An Overview of The BlueGene/L Supercomputer. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, CD ROM, 2002.

[23] O. Lysne, T. Skeie, and T. Waadeland: One-Fault Tolerance and Beyond in Wormhole Routed Meshes. In *Microprocessors and Microsystems*, 21(7/8):471–480, 1998.

Edited by: Sabri Pllana, Siegfried Benkner
Received: June 16, 2008
Accepted: July 28, 2008

Scalable Computing: Practice and Experience
Volume 9, Number 3, pp. 179–184. http://www.scpe.org

ISSN 1895-1767
© 2008 SCPE

SINGLE-PASS LIST PARTITIONING

LEONOR FRIAS[*], JOHANNES SINGLER[†], AND PETER SANDERS[†]

Abstract. Parallel algorithms divide computation among several threads. In many cases, the input must also be divided. Consider an input consisting of a linear sequence of elements whose length is unknown a priori. We can evenly divide it naïvely by either traversing it twice (first determine length, then divide) or by using linear additional memory to hold an array of pointers to the elements. Instead, we propose an algorithm that divides a linear sequence into p parts of similar length traversing the sequence only once, and using sub-linear additional space. The experiments show that our list partitioning algorithm is effective and fast in practice.

Key words: parallel processing, sequences, algorithmic libraries

1. Introduction. An algorithm is a well-defined computational procedure that takes input values and produces output values. A parallel algorithm divides computation among several threads. For data-parallel algorithms, the input must be also divided into (independent) parts of similar size, so that parallel computation is effective. Most parallel algorithm descriptions disregard how the input is actually divided (or assume that the input can be divided by index computations). If we want to use such algorithms in practice, we have to deal with sequences that are non-trivial to partition.

In particular, we focus on the algorithms in the Standard Template Library (STL), which is a part of the C++ programming language [1]. In this setting, the input consists of a sequence of elements, given as a pair of iterators. The standard requires these to satisfy only the forward iterator concept. The only feasible operations for a forward iterator are dereferencing, and advancing to the next element. Thus, a forward iterator sequence is a linear sequence of unknown length (e. g. a sequence implemented as a singly-linked list). Traversing it is inherently sequential. Even if the sequence comes with an associated length variable, this information is lost when passing iterators, as required by most STL algorithms. Also, keeping the length up to date is inefficient for operations like splitting one list into two, at a known iterator. This is why `std::list`, for example, does not guarantee the calculation of `size()` to take only constant time.

However, the speedup of a parallelized program is limited by the sequential portion, according to Amdahl's law. Hence, making the sequential partition before the paralleling processing as fast as possible, is of utmost importance.

Given that the length of the sequence is unknown, one could think of first traversing the sequence to determine its length, and then traversing it a second time to actually divide the sequence. We call this algorithm TraverseTwice. However, traversing the sequence can be expensive, so we do not want to pay for it twice. The elements can be spread in memory cache-unfriendly, and/or calculating the next element may be costly. To avoid this, one could also think of using a dynamic array, storing the pointers to the elements there, effectively converting the sequence to a randomly accessible one. We call this algorithm PointerArray. However, this is very costly in terms of additional space. We subsume both algorithms as the *trivial* solutions.

In this paper, we present an algorithm that divides such a linearly traversable sequence into p parts of similar length using only a little additional space, and accessing each element exactly once. In the next section, we first formally define the problem, called *list partitioning*. Then, we present our single-pass list partitioning algorithm. Next, we present some experiments on list partitioning: we evaluate the algorithms by themselves as well as their impact in parallel performance. Finally, we sum up the results.

2. Problem Definition. A linearly traversable sequence of unknown length n, given by two forward iterators, is to be divided into p parts of almost equal length. Let the ratio r be the quotient of the length of the longest part and the length of the shortest part. It is a good quality measure for the partitioning, since it correlates to the efficiency of processing the parts in parallel, given that processing time is proportional to a

[*]Universitat Politècnica de Catalunya, Dep. de Llenguatges i Sistemes Informàtics. Jordi Girona Salgado, 1-3. 08034 Barcelona, Spain. lfrias@lsi.upc.edu Supported by ALINEX project (TIN2005-05446) and by grants number 2005FI 00856 and 2006BE-2 00016 of the *Agència de Gestió d'Ajuts Universitaris i de Recerca* with funds of the European Social Fund. This work was partially done while visiting Universität Karlsruhe.

[†]Universität Karlsruhe, Institut für Theoretische Informatik, ITI Sanders. 76128 Karlsruhe, Germany.
{singler, sanders}@ira.uka.de Partially supported by DFG grant SA 933/3-1.

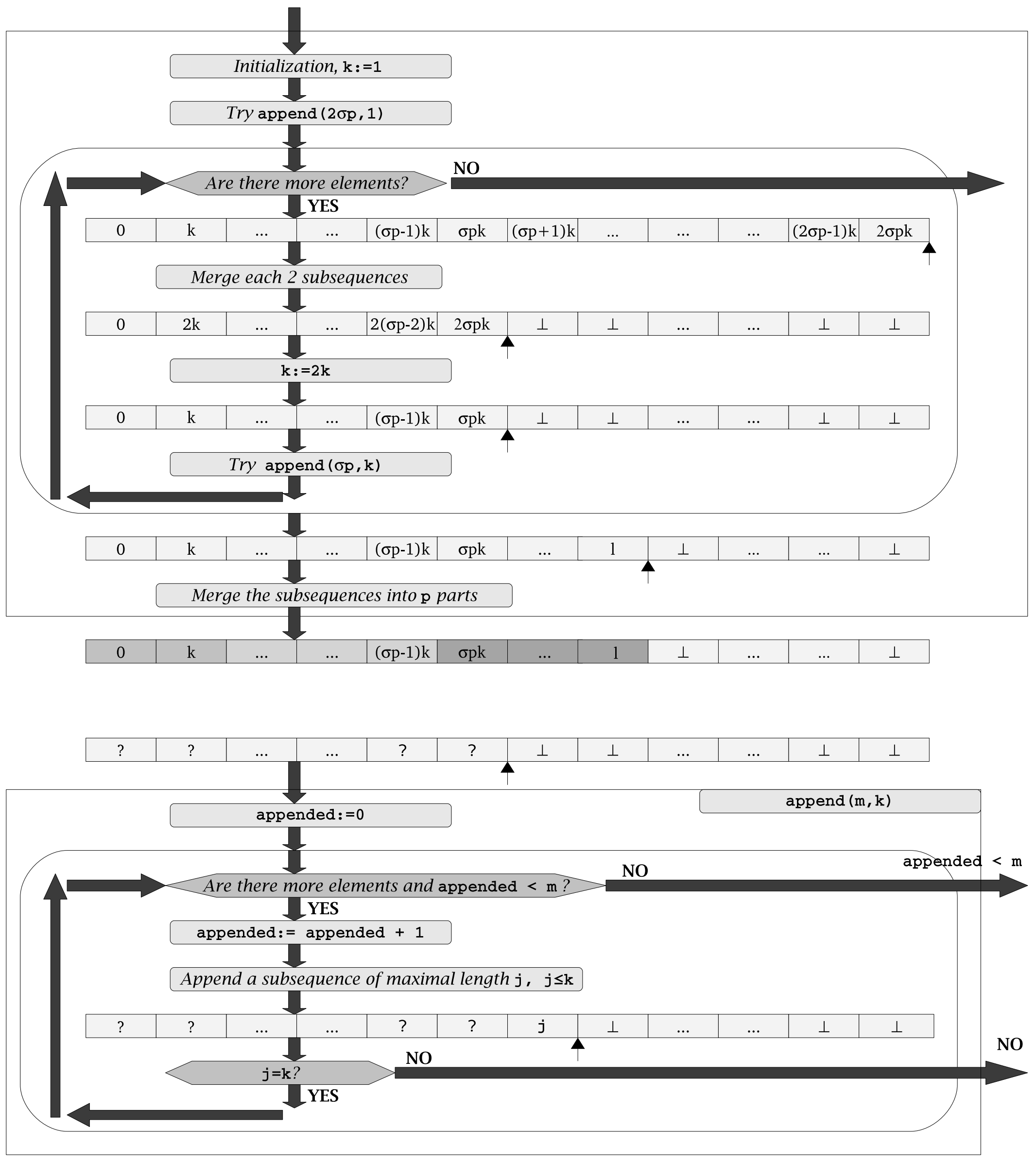

Initialization, k:=1
Try append(2σp,1)
Are there more elements?
NO
YES
0 k ... ... (σp-1)k σpk (σp+1)k ... ... (2σp-1)k 2σpk
Merge each 2 subsequences
0 2k ... ... 2(σp-2)k 2σpk ⊥ ⊥ ... ... ⊥ ⊥
k:=2k
0 k ... ... (σp-1)k σpk ⊥ ⊥ ... ... ⊥ ⊥
Try append(σp,k)
0 k ... ... (σp-1)k σpk ... l ⊥ ... ... ⊥
Merge the subsequences into p parts
0 k ... ... (σp-1)k σpk ... l ⊥ ... ... ⊥
? ? ... ... ? ? ⊥ ⊥ ... ... ⊥ ⊥
append(m,k)
appended:=0
Are there more elements and appended < m ?
NO
appended < m
YES
appended:= appended + 1
Append a subsequence of maximal length j, j≤k
? ? ... ... ? ? j ⊥ ... ... ⊥ ⊥
j=k?
NO
NO
YES

FIG. 2.1. *Basic* SINGLEPASS *list partitioning algorithm scheme.*

part's length. Thus, to guarantee good efficiency, r should be upper-bounded by a constant R at any time, only depending on a tuning parameter, namely the oversampling factor $\sigma \in \mathbb{N} \setminus \{0\}$.

W. l. o. g. we assume that the input sequence has length at least σp, i. e. $n \geq \sigma p$. Otherwise, if $p \leq n$, we can lower σ down so that $\sigma p \leq n$. If $p > n$, we reduce p to n to avoid that any part is empty (and therefore, $r = \infty$), which would not be sensible for our purposes.

Actually, we can consider this an online problem because the input is given one element at a time, without information about the whole problem. Thus, we can define a competitive ratio between the optimal offline algorithm and our online algorithm. For the optimal offline algorithm, the difference in part lengths is at most 1, which gives a ratio $r_{\mathrm{OPT}} = \lceil n/p \rceil / \lfloor n/p \rfloor \overset{n\to\infty}{\to} 1$.

3. The SinglePass Algorithm.

Let L be a forward linearly traversable input sequence (e. g. a linked list). Our single-pass algorithm, denoted SINGLEPASS, keeps a sequence of boundaries $S[0 \dots p]$, where $[S[i], S[i+1])$ defines the i^{th} subsequence of L. Inserting a subsequence into S means storing its boundaries in the appropriate places. A boundary is identified by its rank in L.

The basic SinglePass algorithm works as follows:

1. Let $k := 1$, $S := \{\}$.
2. Iteratively append to S at most $2\sigma p$ 1-element consecutive subsequences from L.
   $S = \{0, 1, 2, \dots, 2\sigma p\}$
3. While L has more elements do:
   Invariant: $|S| := 2\sigma p$, $S[i+1] - S[i] = k$
   (a) Merge each two consecutive subsequences into one subsequence.
       $S[0, 1, 2, \dots, \sigma p] := S[0, 2, 4, \dots, 2\sigma p]$
       This results in σp subsequences of length $2k$.
   (b) Let $k := 2k$.
   (c) Iteratively append to S at most σp consecutive subsequences of length k from L.
       $S := \{0, k, \dots \sigma pk, (\sigma p + 1)k, (\sigma p + 2)k, \dots, l\}$,
       $\sigma pk < l \leq 2\sigma pk$
       If L runs empty prematurely, the last subsequence is shorter than k.
4. The $\sigma p \leq |S| \leq 2\sigma p$ subsequences are divided into p parts of similar lengths as follows. The $|S| \bmod p$ rightmost parts are formed by merging $\lceil |S|/p \rceil$ consecutive subsequences each, from the right end. The remaining $p - (|S| \bmod p)$ leftmost parts are formed by merging $\lfloor |S|/p \rfloor$ consecutive subsequences each, from the left end.

The algorithm (visualized in Figure 2.1) takes special care of the rightmost subsequence E, which might be shorter than the others, i. e. $|E| \leq k$. Let T be the part containing E, there is no part that consists of more subsequences than T. So, if exactly one part is longer than all the others (i. e. $|S| \bmod p = 1$), this is specifically T. In this case, T differs from the other parts in $|E|$ elements. As a whole, the algorithm guarantees that in the worst-case, two parts differ at most in one complete subsequence (i. e. in at most k elements).

The basic SinglePass algorithm needs $\Theta(\sigma p)$ additional space to store S. The time complexity is $\Theta(n + \sigma p \log n)$. This is proved as follows. We need to traverse the whole sequence, taking $\Theta(n)$ time. In addition, step 3 visits $\Theta(\sigma p)$ elements of S in $\Theta(\log n)$ iterations.

The worst case ratio r is bounded by $\frac{\sigma+1}{\sigma}$. The worst case occurs when just one complete subsequence was appended after reducing the list. W. l. o. g., to analyze the average ratio, we consider only complete subsequences, therefore $\sigma p \leq |S| < 2\sigma p$. The average ratio is upper-bounded by

$$\begin{aligned}
\mathbb{E}r &< \frac{1}{\sigma p} \sum_{\ell=\sigma p}^{2\sigma p-1} \frac{\lceil \ell/p \rceil}{\lfloor \ell/p \rfloor} = \frac{1}{\sigma p} \left(\sigma + \sum_{\ell=\sigma p, p\nmid \ell}^{2\sigma p-1} \frac{\lceil \ell/p \rceil}{\lfloor \ell/p \rfloor} \right) \\
&= \frac{1}{\sigma p} \left(\sigma + (p-1) \sum_{\ell=0}^{\sigma-1} \frac{\sigma+\ell+1}{\sigma+\ell} \right) = \frac{1}{\sigma p} \left(\sigma p + (p-1) \sum_{\ell=0}^{\sigma-1} \frac{1}{\sigma+\ell} \right) \\
&= 1 + \frac{1}{\sigma p} \left((p-1) \sum_{\ell=0}^{\sigma-1} \frac{1}{\sigma+\ell} \right) = 1 + \frac{1}{\sigma p} \left((p-1)(\Psi(2o) - \Psi(\sigma)) \right) \\
&\approx 1 + \frac{1}{\sigma p} \left((p-1)(\ln(2o) - \ln(\sigma)) \right) = 1 + \frac{1}{\sigma p} \left((p-1) \ln(2) \right).
\end{aligned}$$

E. g., for $\sigma = 10$ and $p = 32$, the longest subsequence is at most 10% longer than the shortest one, expectedly 7% longer.

A generalization of this algorithm performs step 3a and 3b only every m^{th} loop iteration. In the remaining iterations of the main loop, S is doubled in size, so that space for additional subsequences is needed. This is equivalent to increasing the oversampling factor to σn^γ with $\gamma = 1 - 1/m$.

The generalized SinglePass algorithm needs as many iterations of Step 3 as the basic algorithm, i. e. $\Theta(\log n)$ iterations. The additional space increases, but sub-linearly, growing with $O(\sigma p n^\gamma)$. The time complexity of this algorithm is $\Theta(n + \sigma p(n^\gamma + \log n))$.

In the worst case, the longest sequence and the shortest sequence have length $(n^\gamma + 1)k$ and $(n^\gamma)k$, respectively. It holds $\sigma p n^\gamma k \approx n$, so $k \approx \frac{n^{1/m}}{\sigma p}$. Subsuming this, the lengths of the subsequences do at most differ by $\frac{n^{1/m}}{\sigma p}$ elements, i. e. the difference decreases relatively to n, as n grows. Therefore, the bound for r also converges to 1.

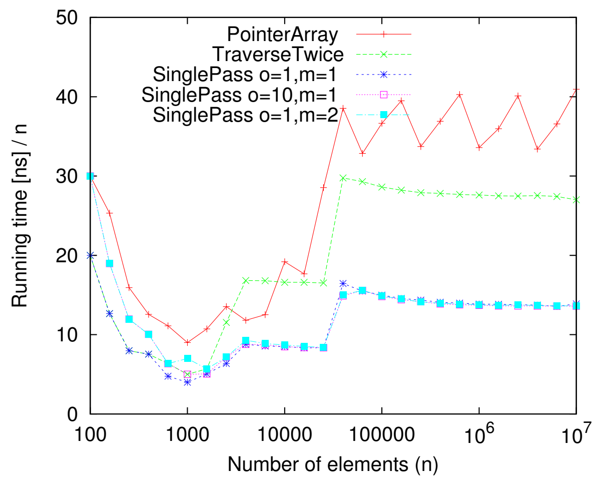


FIG. 4.1. *Running times of the list partitioning algorithms for $p = 4$.*

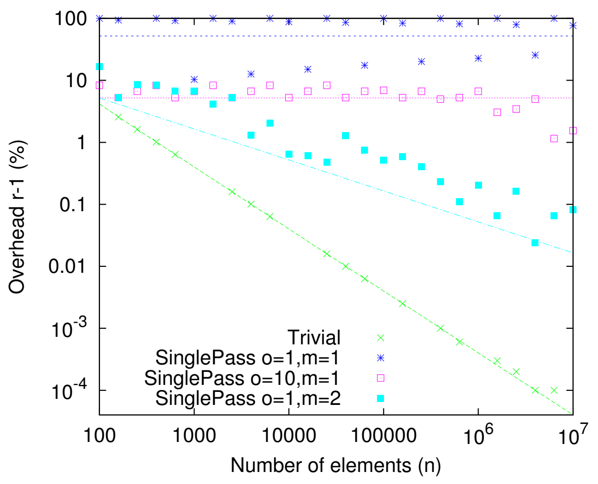


FIG. 4.2. *Quality of the list partitioning algorithms for $p = 4$. We show the worst-case overhead ratio $h = r - 1$, as well as its expectancy. The results are in %. Note that the missing points are actually 0.*

Generally speaking, the choice of m trades of time and space versus solution quality. The larger m, the more memory and time is used, but r becomes better. This is the same effect as would be caused by a dynamically growing σ. Choosing $m = 2$ appears to be a good compromise.

4. Experiments. We have implemented our SINGLEPASS algorithm in the general form, so it subsumes the two variants. We have implemented it in C++ and we have included it into the MCSTL [2]. MCSTL stands for Multi-Core Standard Template Library and is a parallel implementation of the standard C++ library. Besides, we have implemented the two naïve algorithms, namely TRAVERSETWICE and POINTERARRAY algorithms. Dynamic arrays have been implemented using the STL `vector` class.

We have performed two kinds of experiments. First, we present the evaluation of the list partitioning algorithms isolatedly. Then, we present the impact of the list partitioning algorithm as part of two parallelized STL algorithms.

4.1. Comparison of List Partitioning Algorithms. We have compared all the algorithms both measuring the running time as well as the quality of the results. Concerning quality, we have computed both the worst-case ratio r and its expectancy. For a better plot reading, we have rescaled these results using the *overhead* ratio h. h is defined from the ratio r as $h = r - 1$. It must be noted that the actual quality of the results is deterministic with respect to a problem size. That is, the quality of our solution does not depend on the specific input data but only on its size.

Setup.. We have tested our program on an AMD Opteron 270 (2.0 GHz, 1 MB L2 cache). We have used GCC 4.2.0 as well as its libstdc++ implementation, compiling with optimization (`-O3`).

Parameters for Testing. We have repeated our experiments at least 10 times. The focus is on results for $p = 4$. Recall that as p grows, r becomes smaller.

For SINGLEPASS, there are the following parameter combinations: *1)* ($o = 1$, $m = 1$), *2)* ($o = 10$, $m = 1$) and *3)* ($o = 1$, $m = 2$). Therefore, it uses $\Theta(p)$, $\Theta(10p)$, and $\Theta(\sqrt{np})$ additional space, respectively.

Results. Figure 4.1 summarizes the performance results, and Figure 4.2 the quality results. We see that the performance of the SINGLEPASS algorithm is very good. In particular, it takes only half the time compared to the TRAVERSETWICE algorithm, and even less compared to the POINTERARRAY algorithm. That is, we can divide a sequence into parts using virtually the same time as for only traversing it once. Further, the varying running times for POINTERARRAY must be due to the amortization of the vector doubling cost (i. e. depending on how much of the allocated memory has been actually used by the vector).

The quality of the solution for our algorithm improves with the amount of additional space allowed for the auxiliary sequence S (i. e. increased o or m). The simplest variant ($o = 1$, $m = 1$) has a worst-case ratio of 2 and an average ratio of 1.5. In addition, the overhead $r - 1$ is divided by o (in our case, $o = 10$). When setting

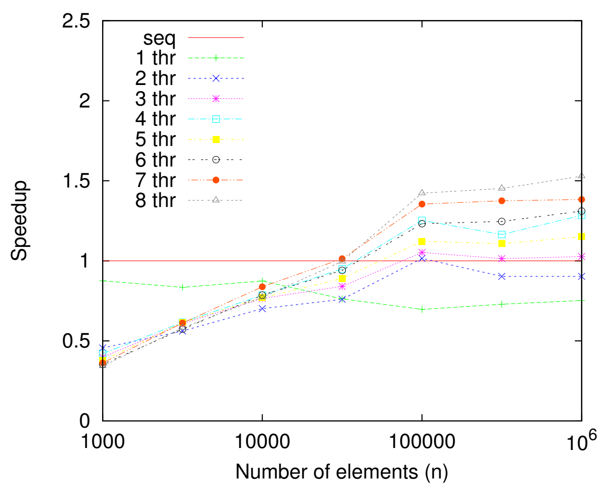


FIG. 4.3. *Speedup for STL list sort using* TRAVERSETWICE *partitioning.*

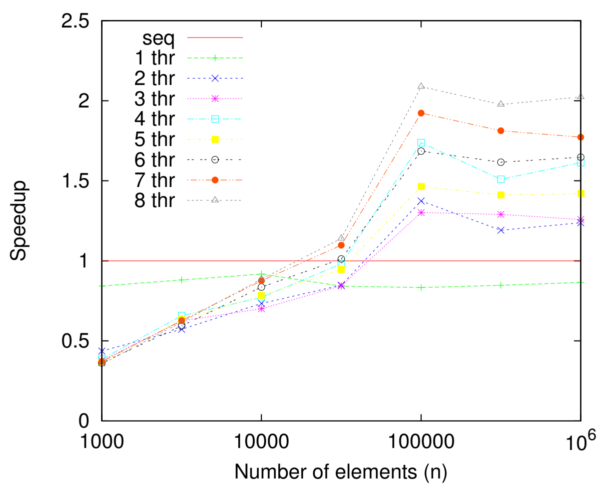


FIG. 4.4. *Speedup for STL list sort using* SINGLEPASS *partitioning.*

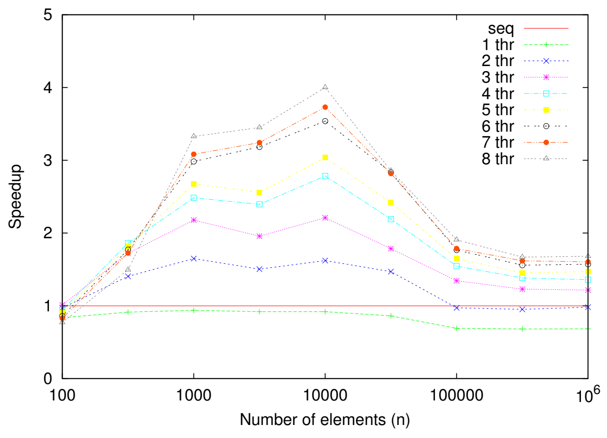


FIG. 4.5. *Speedup for* `accumulate` *using* TRAVERSETWICE *partitioning.*

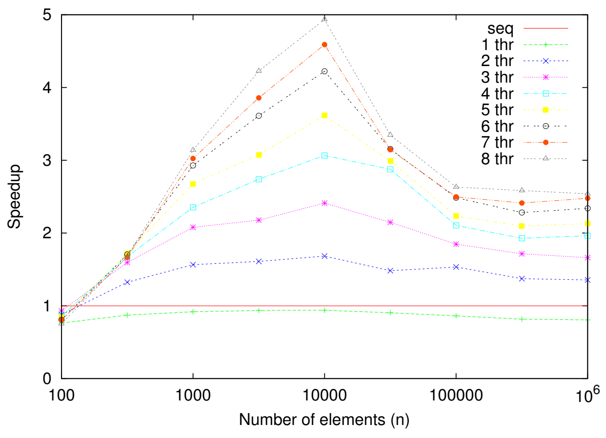


FIG. 4.6. *Speedup for* `accumulate` *using* SINGLEPASS *partitioning.*

m to 2, our algorithm behaves very well, converging to zero overhead. The average overhead ratio decreases exponentially with the input size n. Note that the (optimal) worst-case ratio achieved by the naïve algorithms also decreases exponentially, even faster.

4.2. Parallelization Using List Partitioning. After having evaluated the isolated performance for the list partitioning algorithms, we will investigate their impact in their intended domain, data-parallel algorithms.

Two interesting examples are `accumulate` and `list::sort` from the STL.
`accumulate` computes the sum (or some other reduction based on an associative binary operation) of a sequence of elements, starting with some initial value. Here, we consider the case of a sequence with forward iterators as input. `list::sort` stably sorts a doubly-linked list, whose size is not stored (in favor of fast splice operations). Its implementation is typically based on mergesort[1]. In the following, we first present parallel implementations using list partitioning for both algorithms. Then, we evaluate the impact in performance of different list partitioning algorithms.

Parallel Implementations. Parallelizing `accumulate` is straightforward. Let the desired number of threads be p. The list is partitioned into p parts, using one of the described algorithms. In parallel, each thread accumulates its part. After that, the p results are combined sequentially.

[1]Note that mergesort can be implemented without explicitly splitting the sequence in the middle by using a bottom-up approach.

For the list sort, we partition the list as above and split it into the respective sublists. These are sorted by one thread each, in parallel. Recursive tree-shaped merging combines the results in $\log_2 p$ steps, each merge per se being done sequentially.

Parameters for Testing. We have repeated our experiments at least 10 times and report the average running time.

For SinglePass we have used the parameters $o = 1$ and $m = 2$. Deduced from the results in Section 4.1, this choice produces a high quality partitioning at a low overhead in running time and additional space. The data type for `list::sort` is a 16-byte element containing an 8-byte integer key. The values are randomly generated. As use case for accumulate, we approximately multiply double-precision floating-point numbers by summing up their logarithms.

Setup. We have run these experiments on an dual-socket AMD Opteron 2350 (2×4 cores, 2.0 GHz, 2×2 MB L3 cache).

Results. Figures 4.3 and 4.4 show the speedup for parallel `list::sort` using the TraverseTwice and SinglePass list partitioning algorithms, respectively. The PointerArray variant is not compared here since it has a linear space overhead.

In both cases shown, the achieved speedups in absolute terms are not particularly good. This is probably mostly due to the bad collective memory bandwidth caused by the random accesses to traverse the more and more scattered sublists. Nonetheless, for a large number of elements, the SinglePass list partitioning algorithm is significantly better than TraverseTwice because the sequential fraction is sped up by a factor of 2, and the parts are of very similar size.

Figures 4.5 and 4.6 show the speedup for parallel `accumulate` using the TraverseTwice and SinglePass list partitioning algorithms, respectively.

In this case, the achieved speedups in absolute terms are better because summing the logarithm of floating-point numbers is quite compute-intensive. Again, the performance using SinglePass is much better than with TraverseTwice, in particular for a large number of elements.

Overall, for large inputs, SinglePass obtains much better speedups than TraverseTwice both for `list::sort` and `accumulate`.

5. Conclusions.

We have presented a simple though non-trivial algorithm to solve the list partitioning problem using only one traversal and sub-linear additional space. Our algorithm divides a linearly traversable sequence of unknown length n in time $\Theta(n + \sigma p(n^{1-1/m} + \log n))$ using $O(\sigma p n^{1-1/m})$ additional space. σ and m are tuning parameters of the algorithm.

Our experiments have shown that our algorithm is very efficient in practice. It takes almost the same time as if the list was just traversed, without any processing. Besides, very high quality solutions can be obtained. The larger m, the better the quality, trading off memory. If $m = 1$, the worst-case overhead ratio 1 is divided by the oversampling factor σ. If $m > 1$, the worst-case overhead ratio decreases exponentially with n.

Therefore, for large input instances and in most practical situations, processing perfectly equal parts and almost equal parts should take about the same time, because the time to process each of the parts fluctuates in the same order of magnitude. In addition to this, our approach computes the partitioning twice as fast as the naïve approach.

As a result, using our list partitioning algorithm as a substep of parallel algorithms, the overall performance is significantly improved compared to a naïve implementation.

REFERENCES

[1] *The C++ Standard (ISO 14882:2003)*, John Wiley & Sons, Ltd, 2003.
[2] J. Singler, P. Sanders, and F. Putze, *The Multi-Core Standard Template Library*, in Euro-Par 2007: Parallel Processing, vol. 4641 of LNCS, Springer Verlag, pp. 682–694.

Edited by: Sabri Pllana, Siegfried Benkner
Received: June 16, 2008
Accepted: July 28, 2008

Scalable Computing: Practice and Experience
Volume 9, Number 3, pp. 185–196. http://www.scpe.org
ISSN 1895-1767
© 2008 SCPE

EFFICIENT IMPLEMENTATION OF WIMAX PHYSICAL LAYER ON MULTI-CORE ARCHITECTURES WITH DYNAMICALLY RECONFIGURABLE PROCESSORS

WEI HAN, YING YI, MARK MUIR, IOANNIS NOUSIAS, TUGHRUL ARSLAN, AHMET T. ERDOGAN

Abstract. Wireless internet access technologies have significant market potential, especially the WiMAX protocol which can offer data rate of tens of Mbps. A significant demand for embedded high performance WiMAX solutions is forcing designers to seek single-chip multiprocessor or multi-core systems that offer competitive advantages in terms of all performance metrics, such as speed, power and area. Through the provision of a degree of flexibility similar to that of a DSP and performance and power consumption advantages approaching that of an ASIC, emerging dynamically reconfigurable processors are proving to be strong candidates for future high performance multi-core processor systems. This paper presents several new single-chip multi-core architectures, based on newly emerging dynamically reconfigurable processor cores, for the WiMAX physical layer. A simulation platform is proposed in order to explore and implement various multi-core solutions combining different memory architectures and task partitioning schemes. The paper describes the architectures, the simulation environment, several task partitioning methods and demonstrates that up to 4.9x speedup can be achieved by employing five dynamically reconfigurable processor cores each having individual local memory units.

Key words: multi-core, WiMAX, dynamically reconfigurable, SystemC

1. Introduction. Since the advent of the internet, people have come to depend on it more and more. There is an increasing desire to access it at any time from anywhere. Obviously, cable based internet cannot satisfy this demand. The wireless internet protocol was born in 1997 when IEEE 802.11 [1] was released. In the last decade, more standards have emerged with higher and higher data rates as well as longer and longer ranges. The IEEE 802.16 standard is a wireless broadband access standard which helps make the vision of pervasive connectivity a reality [2]. WiMAX (Worldwide Interoperability for Microwave Access) is its commercial name. WiMAX technology can provide up to tens of Mbps symmetric bandwidth over many kilometers. This gives WiMAX a significant advantage over other alternatives like Wi-Fi and DSL. The physical layer of WiMAX includes both downlink (channel coding and IFFT) and uplink (FFT, modulation and decoding) data processing, as shown in Fig. 1.1. WiMAX applications demand high performance, strict low power, and in-field reprogrammability to follow the evolving standard. Recently, some solutions have been proposed for industry targeting WiMAX that are based on multi-core processor systems, such as [3, 4]. Ever since the first effort to design a parallel machine (called SOLOMON) failed in 1958 [5], the parallel computing paradigm has been improved dramatically. Within the past half century, many advanced technologies have been created and became popular, such as single instruction multiple data (SIMD), symmetric multiprocessing (SMP) and multithreading. The rapid development and demand for parallel architectures are driven by application trends, semiconductor technology trends, and microprocessor and system design trends [6]. Again compelled by those trends, parallel architectures are applied to smaller and smaller computer systems, from the original mainframes, to servers, and to PCs, and now even to embedded systems. Twenty years ago, an 8-bit microcontroller could satisfy most of the requirements of embedded systems. But now, even an advanced 32-bit reduced instruction set computer (RISC) processor may not be able to realize a complicated embedded application. Moreover, as more and more complex applications are mapped onto embedded systems, and the concern about power consumption and heat dissipation increases, there is a need to distribute the processing load of large embedded programs over multiple processors. A multi-core processor gains the advantage over multi-chip SMP in terms of all performance metrics, due to the shorter distances the signals between processors have to travel. Here a multi-core processor (or multi-core) is defined as a single chip combining two or more independent general purpose processor or DSP cores.

Today's common general-purpose processors or DSPs cannot satisfy all the features required by current and future embedded systems, including higher performance, higher power efficiency and higher flexibility. Filling the gap between the high flexibility of DSPs and the high performance of ASICs, dynamically reconfigurable (DR) processors offer an alternative solution for embedded systems. This paper extends our work in [27] and presents several new single-chip multi-core architectures, based on newly emerging dynamically reconfigurable processor cores, for the WiMAX physical layer. To address this, we have proposed a flexible simulation platform that enables the analysis and investigation of multi-processor systems based on coarse-grained DR processors—an area that as yet has been little explored. The WiMAX 802.16-2004 physical layer was ported on this platform.

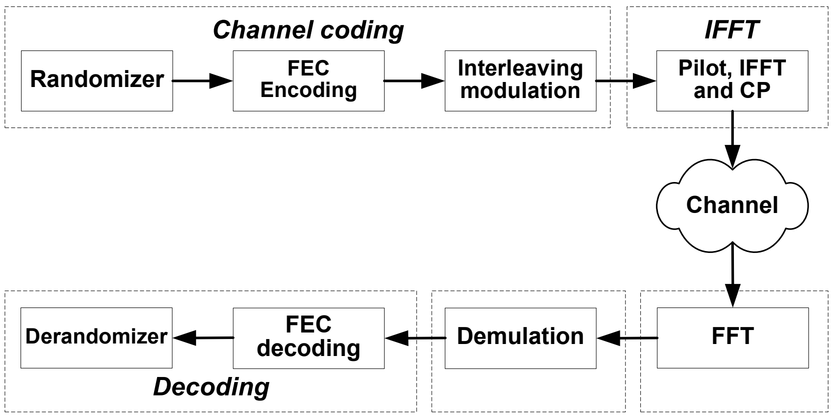


Fig. 1.1. *WiMAX physical layer processing chain*

The rest of this paper is organized as follows: section 2 reviews related research work on both WiMAX implementations and multi-core processor solutions for embedded systems. Section 3 provides a brief overview of the target dynamically reconfigurable processor and introduces the underlying multi-core processor architecture within the platform. Section 4 presents a SystemC TLM based simulator which models this multi-core processor architecture. Section 5 describes how to map multiple tasks onto the multi-core processor architecture and several different task partitioning scenarios for efficiently implementing WiMAX physical layer on the multi-core processor simulator, the results of which are shown in section 6. Finally, conclusions are given in section 7.

2. Related Work.

2.1. WiMAX Implementations. Many market-leading companies today provide products based on silicon implementations of WiMAX standards. Such products include ASIC, DSP or network processor technologies. For example, Wavesat DM256 [11] is an ASIC solution, and the Intel NetStructure WiMAX Baseband Card is based on the Intel IXP2350 network processor [12], while both Freescale MSC8126 DSP [3] and Picochip PC203 [4] use the multi-core DSP approach. In [13], a multitasked version of the WiMAX physical layer has been mapped onto a single DR architecture with a real-time operating system (RTOS) - μC/OS - II.

2.2. Chip-level Multiprocessor in the Embedded Field. System architects today acknowledge that higher clock rates, deeper pipelining, and increasing instruction-level parallelism (ILP) cannot be relied on any more in order to increase performance. This is due to overheads introduced by such techniques. For this reason, industry has started looking at multi-core processor solutions. Besides multi-core DSPs mentioned above, other multi-core processor products include Cell Broadband Engine Architecture [18], ARM11 MPCore [20], intellaSys SEAforth-24A [21], and Ambric Am2000 family [19]. Few multi-core processor projects are based on coarse-grained DR processors: One such example being the work on developing a multi-core processor based on the Montium Tile Processor (TP) from RECORE [14]. However, to the best of our knowledge, currently Montium TP does not support development from high-level languages, and its homogeneous processing part array (PPA) is not efficient in terms of hardware resource occupancy. While our chosen DR processor can provide high performance at low power and area, addressing all the desired requirements for future portable devices. The cell array configuration can be tailored towards different application domains. A series of power saving techniques have been adopted, which include using registers for interim storage, avoiding large multiplexers and multiple-load busses, increasing the program kernel, and using heterogeneous instruction cells.

3. Multiple Reconfigurable Processor Architecture.

3.1. The target Dynamically Reconfigurable Processor. Recently, several new coarse-grained DR architectures have emerged, such as HiveFlex [7], RaPiD [8], and the reconfigurable instruction cell array [9]. In this paper, the target DR processor used within the proposed multi-core processor platform is the DR architecture introduced in [9].This architecture consists of a heterogeneous array of instruction cells interconnected through an island-style mesh fabric, and supports development from high level languages such as C in a manner very similar to conventional microprocessors and DSPs. One of the key features of the chosen architecture is that it is largely indistinguishable from a conventional processor, except for the memory access patterns (for both data and program memory) which can be quite different. This is one of the key issues addressed in this research. It supports operation chaining—the ability to execute both dependent and independent instructions in parallel, which leads to high degree of parallelism. This demands higher data memory bandwidth in order to keep the array fed with data, a demand which has been met by providing a multi-bank memory system that is quite different to that seen in conventional processor based multi-core processor systems. Furthermore, the instruction stream fetch patterns are unusual in that successive iterations of certain loops can be executed following only a single fetch. The instruction words are also quite large compared to those in existing DSPs. ANSI-C programs can be compiled into a sequence of configuration contexts, the contents of which are executed concurrently by the DR processor according to the availability of hardware resources. Formed by a sophisticated scheduling algorithm [10], each context packs together as many chains of operations as possible, and takes a variable number of clock cycles to execute, in order for all chains to complete. Another salient characteristic of the DR processor is that it is tailorable and extensible, and can be customised at design stage according to the application requirements. A number of algorithms have been tested to demonstrate that the target processor can achieve up to 6 times less energy consumption than a DSP such as a TI very long instruction word (VLIW) and 8 times higher throughput than RISC CPUs such as OR32 [9].

3.2. Master-Slave Shared Memory Architecture. In this paper, we propose a master-slave based shared memory architecture for DR processors (shown in Fig. 3.1), targeting WiMAX applications. The shared memory architecture is popular for multiprocessors based on a small number of cores. One of the key aspects of this architecture is that it provides support for DR processors that may issue multiple concurrent memory access requests per cycle. This architecture consists of one master processor core and multiple slave processor cores. The main difference between the master and slaves is that the master core takes charge of the task management. When the system starts up, the master dispatches tasks to slaves by sending the task information through a router. When a slave finishes its current job, it sends an interrupt to the master, requesting a new job. Then the master runs a corresponding interrupt handler to dispatch a new job to that slave. The target DR processor is used for both master and slave cores, since it is suitable for executing both data path and control path programs, as presented in [13], where both a RTOS and WiMAX protocol run concurrently on a single DR architecture. To address the parallel memory access requirement of DR processors, this multi-core processor architecture is based on a Harvard architecture where each core owns a program memory and all processor cores share a multi-bank based data memory. Besides the shared data memory, each DR processor core can have its own local private multi-bank data memory which cannot be addressed by other processors. By having much smaller access latency than the shared memory, due to the adjacency to the processor and smaller depth, the private memory alleviates the conflict in accessing the shared memory, and improves the data locality, and therefore improves the throughput. Each data memory bank has an arbiter which arbitrates the memory accesses to the bank from different processor cores or different memory ports of the same DR processor core, which may contain multiple memory access ports. All shared data memory banks are connected to all processor cores through switch bars. According to the number of banks and the referenced address, each memory access request to the shared data memory from a memory port of a processor core is routed through switches to the proper arbiters. Each processor core has one local interrupt controller. An interrupt channel array is provided for communication transactions between interrupt resources and interrupt controllers.

3.3. Proposed Architectural Extensions. Besides shared and private memory blocks shown in Fig. 3.1, this multiprocessor architecture has been extended and modified by incorporating the shared register file into the system memory architecture in order to speed up data exchange and communication between processors. Multi-port register files are commonly used for VLIW processors which need to access several registers simultaneously. It is well known that the number of ports and the size of the register file affect its energy consumption, access time and area. Most of the previous work on the register file has been related to techniques of reducing access time and power consumption. Some of the authors [22] have studied the hardware complexity of shared register files and proposed distributed schemes as opposed to a central implementation. Similarly, [23, 26] used techniques to split the global micro-architecture into distributed clusters with subsets of the register file and functional units. Multilevel register file organizations [24] have also been introduced to reduce the register files size. All the above mentioned works focus on reducing the number of registers. Conversely, other techniques split the register file into interleaved banks which reduces the total number of ports in each bank, but retain the idea of a centralized architecture [25]. Different from other techniques, we proposed to split the register

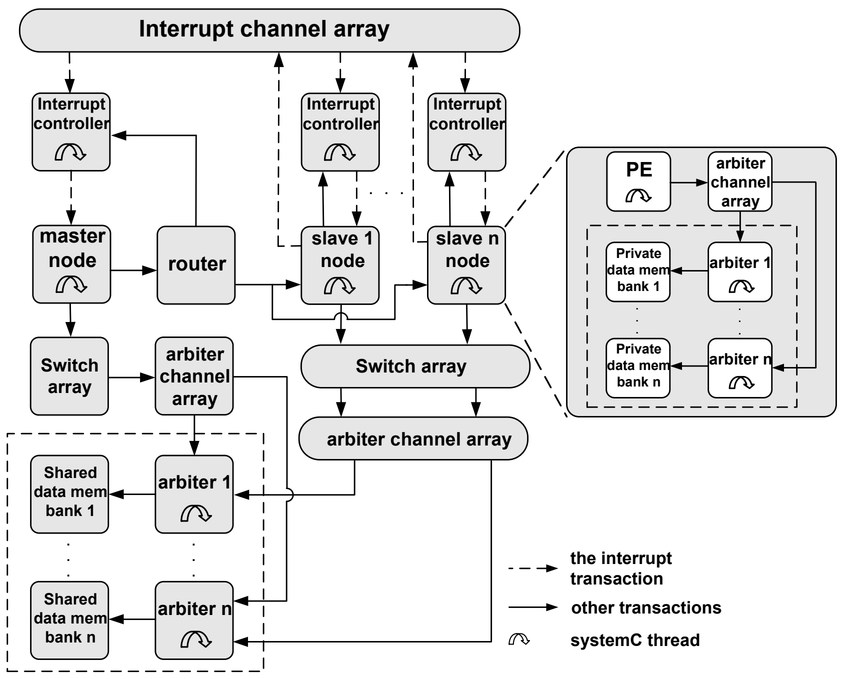


FIG. 3.1. *Master-slave shared data memory architecture*

file into independent banks with a reduced number of ports per bank for reducing the number of register file ports.

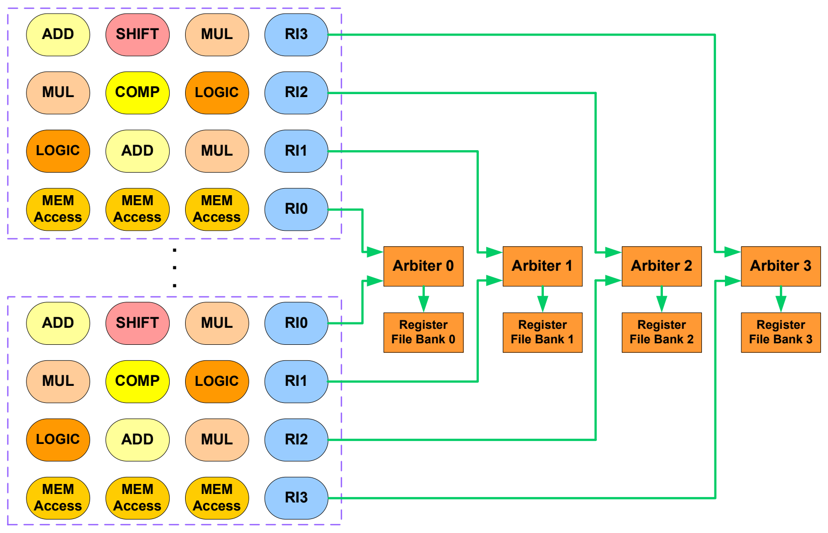


FIG. 3.2. *The shared register file implementation*

The proposed approach employs a multi-bank dual-port register file where each bank consists of a 32x32-bit 1-write-port and 1-read-port data register file. Some custom register file interface cells, which provide access to these banks, are introduced into the target DR processor to support distributed shared register files. The register file interface cells have 32-bit for data write, 32-bit for data read, 5-bit for write address, 5-bit for read address, and 2-bit for configuration which indicates the access model as read-only, write-only or read before write. The number of register file banks can be parameterised at synthesis stage. Fig. 3.2 illustrates a design of a shared 4-banks register file where each bank has an individual arbiter. All instruction cells in the DR processor can be connected to all register interface cells, and each interface cell is only able to connect to one fixed register file bank. For example, register file interface cell RI0 is only connected to bank0 and register file interface cell RI1 can only access bank1. This scheme keeps all possible connections between the instruction cell and the shared register bank with a reduced MPSoC interconnection complexity.

The existing DR processor tool-flow provides full support for the inclusion of both combinatorial and synchronous custom instruction cells through the simulator libraries. The function descriptions of the custom cells such as the interface cells are modeled in C++ via template classes provided by the simulator. A fully automated system generator compiles the standard simulator libraries with the custom cell objects and generates all the required support files and the custom simulator, replacing the standard simulator used in the DR processor tool flow. The shared register file has been synthesised with Cadence Ambit BuildGates using the UMC 0.18um standard CMOS cell library which the DR processor is based on. The power consumption of the synthesized register file has been obtained using Synopsys PrimePower. The generated timing, area and power information is used in the DR processor tool flow for scheduling and simulation purposes.

3.4. Atomic Operation Support. In order to avoid race conditions caused by simultaneous accesses to a shared resource from different processes, in this work the process synchronization was implemented with the use of spinlocks. A spinlock is a lock where a task repeatedly checks the lock (synchronization variable)—a method called busy-waiting. A spinlock allows only one task to access the shared resource protected by the lock at any given time. Implementing a spinlock requires the support of atomic read-modify-write operations where the content of a memory location is guaranteed to be read, modified and written back without intervening operations from other processor cores. In this work, a pair of special instructions—load-linked and store-conditional (i. e. LL/SC) [6] are used to implement atomic operations and therefore the spinlock. The idea behind the LL/SC is that LL loads the synchronization variable into a register, and is immediately followed by an instruction that manipulates the variable. For the spinlock, this instruction adds 1, then SC tries to write the variable back to the memory location if and only if the location has not been written by other processor cores after LL was completed by this processor core [6]. A pseudo-assembly code for the spinlock and unlock algorithms implemented by LL/SC is shown below [6]:

```
Lock: LL reg1, location // LL the variable
BNZ reg1, lock // if locked, try again
SC location, reg2 //SC the variable
BEQZ lock //failed, try LL again

Unlock: WRITE location, 0 //write 0 to location
```

In the proposed multiprocessor architecture, for implementing LL/SC, each data memory arbiter contains a hardware lock flag and a lock address register for each processor. When an LL instruction from one processor reads a synchronization variable, the corresponding lock flag is set and the variable address is stored into the corresponding lock address register. Whenever the referenced address of a write request (either normal write or SC) is matched against the value of the lock address register, the lock flag is reset. A SC instruction enables a check of the lock flag. If it has been reset, meaning that an intervening conflicting write occurred before, SC fails, otherwise, it succeeds.

Obviously, the busy-waiting based spinlock causes more memory access conflicts and more power consumption since the waiting core keeps checking the synchronization variable. In this paper, we have presented an optimized method, called Lock&Signal, which can reduce the memory access competition and therefore the power consumption. The idea is that if the lock required by a processor core is unavailable, the task will go to sleep and the processor core will do nothing but wait. When the lock is released by another process, the releasing process will send a signal to the waiting process, informing it that the lock is now available, through an inter-processor interrupt. Then the waiting process can resume its operation. Therefore, memory access conflicts due to the busy-waiting based spinlock, which may take up a reasonable ratio of the execution time, are eliminated.

4. Multiprocessor Simulator. In this paper, we use SystemC transaction-level-modeling (TLM) [15, 16] to model the proposed multi-core processor architecture. There are a number of existing multiprocessor simulation tools proposed by both the industry and academia [28, 29, 30, 31, 32, 33]. In [28], a multiprocessor enhancement is proposed for SimpleScalar which is a popular simulation tool in the research community and models a MIPS-like architecture. The authors of [29] presented a simulator called Rsim for shared-memory multiprocessors with ILP processors. Both GEMS [30] and RASE [31] simulators interact with Simics, a full system simulator, by feeding timing information to it. Obviously, the simulation speed is dramatically decreased since Simics is involved during the multiprocessor simulation. While SESC simulator [32] uses the instruction streams generated by MINT, an emulator for MIPS architecture, to model chip-level multiprocessors with out-of-order processors. In [33], a simulation environment called MPARM is proposed based on SystemC register transfer level (RTL) and SWARM (software ARM) simulator. However, partially describing the system at lower SystemC abstraction level, MPARM is inevitably slower than a TLM based simulation tool. Moreover, most of these simulation tools are designed for multiprocessors with conventional processor architectures like ARM and MIPS. To the best of our knowledge, there has been little research done for developing TLM based simulation tools for MPSoCs targeting coarse-grained DR processors (e.g., [9]).

TLM is a higher abstraction level than RTL, and is a transaction-based modeling approach which enables concurrent hardware/software co-design for early software development, architecture analysis and functional verification. In TLM, system components are modeled as modules containing a number of concurrent SystemC threads which represent their behavior. The communication among modules is achieved using transactions through abstract channels which implement TLM interfaces. As the kernel of TLM, TLM interfaces can be accessed by SystemC threads through module ports [16]. Fig. 3.1 illustrates how the architecture is implemented using TLM. The master processor core, slave processor cores and arbiters are modeled as initiators which contain SystemC threads initiating transactions to the target components such as memory banks and the router. Meanwhile, an initiator could be a target as well, like the processor cores which also can receive interrupt transactions from the interrupt controllers. Based on run-time communication between the memory system and the DR processors, our multi-core processor simulator maintains the timing accuracy with a rapid architecture analysis. The generated TLM executable, which is called multiple reconfigurable processor simulator (MRPSIM), is parameterizable in terms of the number of slave processor cores, the number of shared memory banks, inter-process communication methods, single core or multi-core model and so on. For example, by setting an option, each DR processor core can have its own local private memory which stores the local data for the associated tasks on the processor core, while the shared data memory is only used for global data and synchronization variables.

MPSIM is a trace-driven simulator. The advantage of the trace-driven simulation over execution-driven simulation is that once the trace has been collected, it can be reused as a constant input to the simulator when the MPSoC architecture varies. However, a difference from other trace-driven simulators - which sacrifice accuracy for performance - MRPSIM can maintain the timing accuracy with a rapid architecture analysis. The idea is that the functionality of the programs and the calculation of static timing are decoupled from dynamic timing simulation. In our simulation approach, the entire simulation timing is separated into static timing and dynamic timing. Static timing represents the time consumed by the combinatorial critical path in each configuration context. It is not affected by the run-time execution. Dynamic timing refers to the time taken by communication instructions (e.g., load and store memory), which can only be determined during at run-time in the multiprocessor simulation, due to competition such as from multiple processors, or multiple memory accesses from one of the processors.

To implement this approach, each program to be executed on MRPSIM should first be executed on a single DR processor simulation model. MRPSIM uses static timing and instruction information from the execution traces provided by the single DR processor simulator to determine the length of time each processor core is occupied between memory accesses. Then, the dynamic delays imposed by memory access, process synchronization and interrupts are obtained from the run-time behaviour of the TLM model. Through this method, MRPSIM can correctly and efficiently simulate the run-time multi-core processor environment, allowing the throughput of the modeled system to be measured. This approach is well suited to programs with simple control flow, such as the WiMAX physical layer.

5. Mapping Methodology and Task Partition.

5.1. Mapping Methodology. For running multiple tasks onto the MRPSIM simulator, the mapping methodology shown in Fig. 5.1 was developed. First, the target application is partitioned into multiple tasks. Inter-task synchronization is done using the optimized Lock&Signal method, explained in Section 3. Together with global data and synchronization variables such as locks, each task is compiled and simulated separately with one DR processor. The resource mix for each processor in the system is allowed to differ, so that it may be tailored to the particular tasks that it is intended to execute. The single DR processor simulator generates an execution trace file for each task, which records the detailed program execution and the static

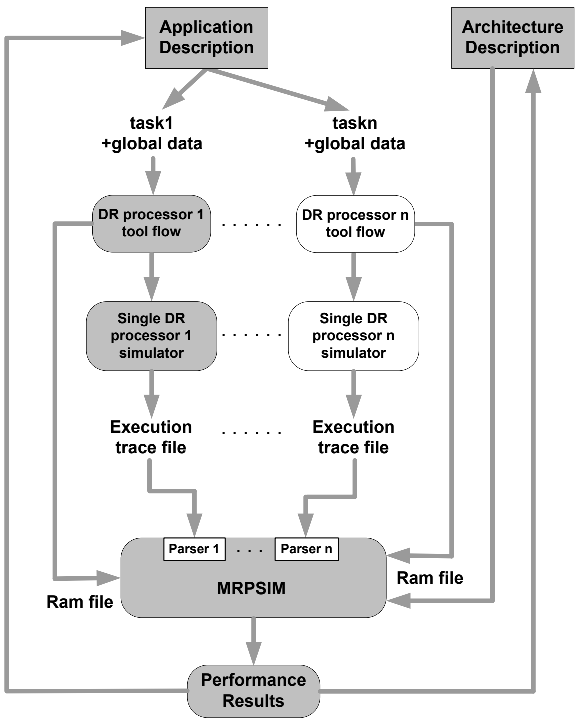


Fig. 5.1. *Mapping methodology*

timing information for configuration contexts. After that, parsers within MRPSIM convert those machine-specific execution trace files into a machine-independent intermediate representation that captures the timing model and flattened program control flow structure. Through this method, the multi-core processor model can support both homogeneous and heterogeneous DR processor based multi-core architectures, or even conventional DSPs. Other inputs of MRPSIM include the RAM files describing the layout of data symbols in the various data memories, and an architecture description file which contains information for both the multi-core processor architecture and the single DR processor. After performing simulations with MRPSIM, generated results are used as feedback to change the design strategy such as the task partitioning and architecture parameters in order to achieve better performance.

5.2. Task Partition Methods. Important issues in multi-processor System-on-Chip (MPSoC) designs are the communication infrastructure and task mapping. We use profiling-driven partition and static mapping solutions. The execution time of a task performed by a single processor can be obtained by software profiling using an instruction set simulator. Task merging and task replication are two main process transformations to improve the application throughput [17]. The WiMAX physical layer application consists of five tasks defined in [13], which are channel coding, IFFT, FFT, demodulation and decoding, as shown in Fig. 1.1, with individual execution times of 60us, 108us, 109us, 28us and 1206us, respectively. The design challenge is to map the application onto an architecture optimized in terms of the system performance and cost. This includes not only the mapping strategy but also architectural design choices such as the number of processor cores and each processor's make up in terms of number of instruction cells and their types.

A simple mapping would assign each task to a processor core, resulting in a multi-core processor architecture with five processor cores. However, this partitioning method would lead to a highly imbalanced workload. The overall throughput of the WiMAX application is limited by the most time-consuming task, which is the decoding task. To improve the load balance, merging and replicating tasks can be employed in the mapping strategy. We employ a second partitioning method which merges the channel coding, IFFT, FFT and demodulation tasks into a single task that sequentially performs the computation of the original tasks. Task merging reduces the number of tasks from five to two, as shown in Fig. 5.2. However, the throughput for this partition is still determined by the most time-consuming task (decoding). To further improve the throughput, task replication can be applied to the decoding task. For example, the decoding task can be replicated to another three DR

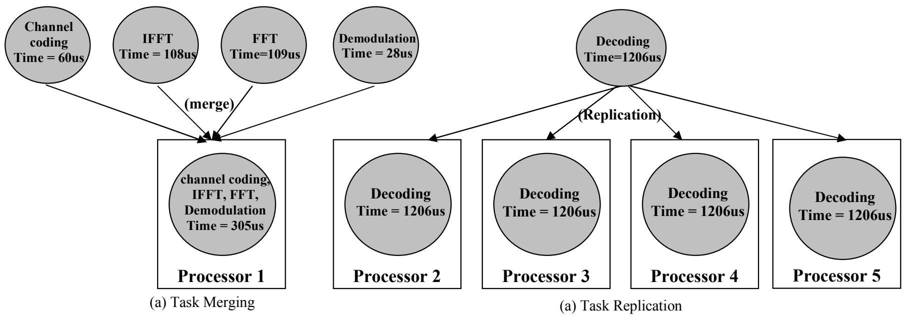


Fig. 5.2. *Task Merging and Task Replication Methods*

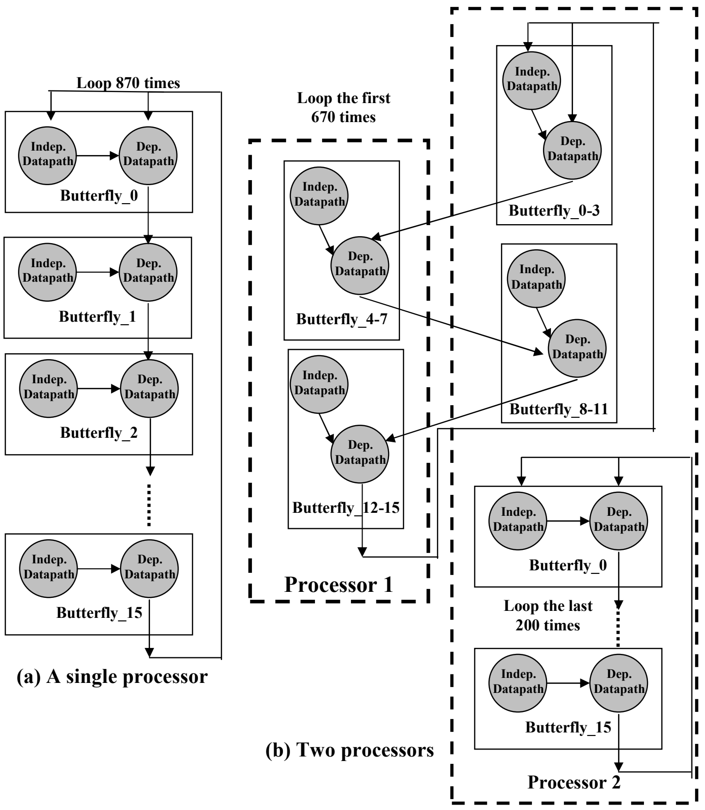


Fig. 5.3. *Loop Partitioning Method*

processor cores, as the execution time of the decoding task is four times that of the merged task. Ideally, the decoding task will complete four parallel instances every 1206us, resulting in a 300% performance improvement. Clearly, task merging and task replication can lead to higher performance solutions. However, task merging requires more instruction memory and task replication requires more global data memory as well as more DR processor cores. The execution time of the decoding task is larger than the execution times of all the other tasks combined. Therefore, with two processors, the best allocation that can result from task merging alone is to place the decoding task on the second processor, and all others on the first processor. However, this still leaves the first processor idle for a significant fraction of the time, resulting in unbalanced workload. In order to balance the workload among the two processor cores, we propose the fourth partitioning method, called loop partitioning. This method uses loop splitting and instruction level partitioning in order to parallelise the resulting code across different DR processor cores. The decoding task contains the most time-consuming loop body with a loop-carried dependence, which means each iteration of a loop depends on values computed in an earlier iteration. This prevents further efficient partitioning at the task level. As shown in Fig. 5.3 (a), the body of the main loop of the decoding task iterates 870 times and includes 16 butterfly operations. A butterfly operation consists of independent and dependent data paths. The independent data paths can be run on different processor cores to increase the instruction level parallelism. Loop splitting is a compiler optimization technique that attempts to simplify a loop or eliminate dependencies by breaking it into multiple loops which have the same bodies but iterate over different contiguous portions of the index range. To make use of this idle time, loop splitting can be used to effectively move some of the work of the decoding task from the second processor onto the first processor, during this idle time. To achieve this, the decoding loop body is written in two ways: one with all the work being done on a single processor, and another where some of the butterflies are moved to another processor, such that they can be performed in parallel, but out of sync to allow for the dependencies to be met. The second processor then runs the first (single-processor) version when the first processor is busy executing the merged tasks, then switches to the second version (dual-processor), allowing the now idle first processor to share in the decoding task. Based on the execution time of the merged task, the loop is split into two parts: the first 670 iterations and the last 200 iterations. The execution time of the last 200 iterations is nearly equal to the merged task. The body of the loop in the first 670 iterations is partitioned across two processor cores, as shown in Fig. 5.3(b). After loop partitioning, the merged task and part of the first 670 iterations are assigned to one processor core, the remaining part of the first 670 iterations and the whole of the last 200 iterations are assigned to the second processor core. The first processor's idle time can be best absorbed in this way by executing the dual-processor version of the decoding task for 670 iterations. The loop partitioning results in a balanced workload; however it may introduce a large number of data exchange between cooperating processor cores and increase communication overhead. If the data exchange happens in the shared memory, the shared memory could become a performance bottleneck of the loop partitioning method due to the frequent additional data communication between the DR processors. Therefore, a feasible optimization method is mapping the frequent read and write shared data to the shared register file for reducing memory access time and memory access conflicts.

6. Results. Several WiMAX multiprocessing scenarios, which combine different task partitioning methods and memory architectures, have been implemented and simulated with MRPSIM. These scenarios are described in Table 6.1. To make fair performance comparisons, all scenarios are executed with 100 OFDM symbols. The clock frequency of the DR processor was set to 500MHz and the shared memory access delay was set to 6ns, the local private memory access delay was set to 2ns, and the shared multi-bank register file access delay is 1ns. Meanwhile, the WiMAX was executed on the single processor simulator with the same simulation parameters except the local private memory and shared register file, for comparison.

Table 6.1
Multiprocessing scenarios

	No. of DR processors	Partitioning methods	Memory mapping
Scenario1	2	Task merging	Shared
Scenario2	2	Task merging	Shared&Private
Scenario3	2	Task merging + Loop partitioning	Shared&Private
Scenario4	2	Task merging + Loop partitioning	Shared&Private&Regfile
Scenario5	5	Task merging + Task replication	Shared
Scenario6	5	Task merging + Task replication	Shared&Private

Figures 6.1 and 6.2 show the speedup and parallel efficiency of WiMAX implemented on multi-core processors with the six different scenarios. The speedup is obtained by dividing the simulation time on single processor by the simulation on multi-core processor. While the parallel efficiency (defined as the speedup divided by the number of processor cores) is used to estimate how efficiently the processors are utilized in solving the problem [6]. Table 6.2 shows the idle ratio of DR processor cores for different scenarios. The idle ratio refers to the ratio of the period when the DR processor core is idle to the overall simulation time. As the results show, the scenarios employing task merging and replication achieve both the highest speedup and the highest parallel efficiency, compared to other task mapping methods. This is mainly because task replication employs more processor cores and dispatches more balanced workload to DR processor cores. In scenarios with task merging and replication, all processor cores are much more efficient with very low idle ratios. The scenarios with local

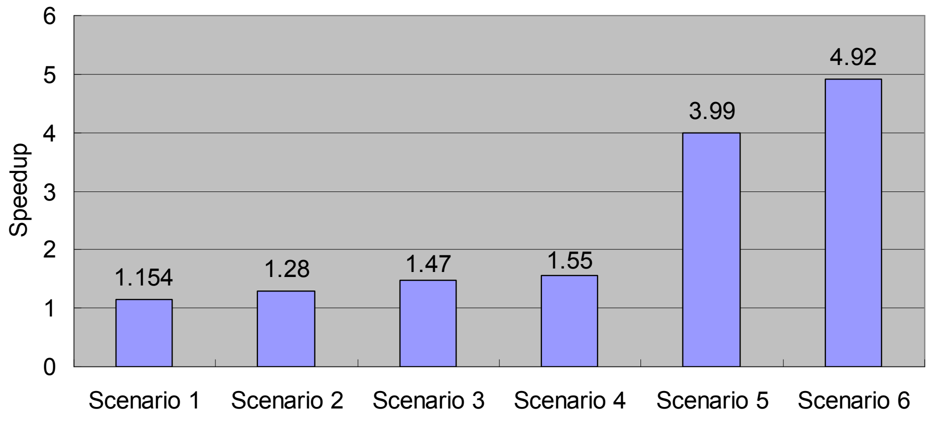


Fig. 6.1. *The speedup of scenarios over a single DR processor*

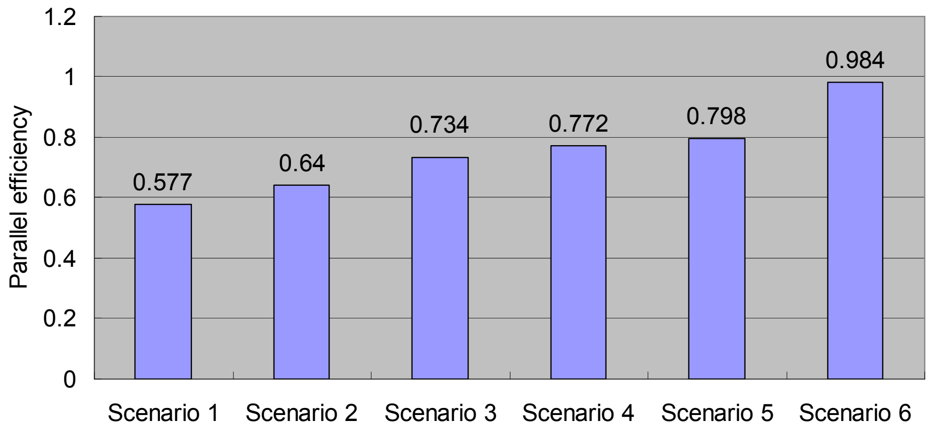


Fig. 6.2. *The parallel efficiency of scenarios*

memory have better throughput compared to their counterparts which have shared data memory only. Scenario 6 with local memory achieved 4.92x speedup and a parallel efficiency of 0.98 with five target DR processor cores. Due to the deeply imbalanced workload, scenarios only using task merging demonstrate very high idle ratios for the processor cores. Based on the multi-core architecture with limited processor cores, scenario 4, which uses task merging and loop level partitioning, has a much lower total idle ratio and better performance compared to scenarios using task merging only. This is due to the improved instruction level parallelism and the efficient multi-core architecture with a shared register file between different processor cores.

Table 6.2
The idle ratio of each PE for individual scenarios

	PE0 (master)	PE1 (Slave0)	PE2 (Slave 1)	PE3 (Slave 2)	PE4 (Slave 3)	Average
Scenario1	73.5%	0.3%	-	-	-	36.9%
Scenario2	75.9%	0.2%	-	-	-	38.1%
Scenario3	12.2%	10.5%	-	-	-	11.4%
Scenario4	14.3%	9.6%	-	-	-	12.0%
Scenario5	3.5%	8.4%	8.3%	8.3%	8.4%	7.2%
Scenario6	7.4%	7.4%	3.7%	3.7%	3.7%	5.2%

7. Conclusion. Several multi-core processor solutions have been proposed for WiMAX physical layer applications, based on coarse-grained dynamically reconfigurable processor cores. Three different task partitioning methods have been applied, and their impact on the system performance has been discussed. A flexible multi-core processor simulation platform, MRPSIM, has been proposed and developed in order to evaluate different solutions which combine different task partitioning strategies and memory architectures. Simulation results have demonstrated the effectiveness of the proposed strategies in terms of speedup and parallel efficiency.

REFERENCES

[1] *IEEE 802.11-1997: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*, 1997.
[2] *IEEE Standard for Local and Metropolitan Area Networks*, IEEE STD 802.16-2004, 2004.
[3] *Implementing WiMAX PHY Processing Using the Freescale MSC8126 Multi-Core DSP*, Freescale Semiconductor WiMAX WP White Paper, April 2004.
[4] *Picochip PC203 produce brief*, Picochip.
[5] G. Wilson, *The History of the Development of Parallel Computing*, `http://ei.cs.vt.edu/ history/Parallel.html`
[6] David Culler, J.P. Singh and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufmann, 2nd edition 1999.
[7] HiveFlex programmable cores, www.siliconhive.com
[8] Carl Ebeling, Chris Fisher, Guanbin Xing, Manyuan Shen, Hui Lui, *Implementing an OFDM Receiver on the RaPiD Reconfigurable Architecture*, IEEE Trans. Computers, Vol. 53, No. 11, Nov. 2004, pp. 1436–1448.
[9] S. Khawam, I. Nousias, M. Milward, Y. Yi, M. Muir and T. Arslan, *The Reconfigurable Instruction Cell Array*, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 16, Issue 1, pp. 75–85, Jan. 2008.
[10] Y. Yi, I. Nousias, M. Milward, S. Khawam, T. Arslan and I. Lindsay, *System-level Scheduling on Instruction Cell Based Reconfigurable Systems*, Design, Automation and Test in Europe (DATE 06), Vol. 1, pp. 1–6, March 2006.
[11] *WiMAX 3.3-3.8GHz Mini-PCI Reference Design Series*, Wavesat, 2007.
[12] *Intel NetStructure WiMAX Baseband Card Product Brief*, Intel Corporation, 2006.
[13] W. Han, I. Nousias, M. Muir, T. Arslan and A. T. Erdogan, *The Design of Multitasking Based Applications on Reconfigurable Instruction Cell Based Architectures*, Int. Conf. on Field Programmable Logic and Applications (FPL 2007), pp. 447–452, Aug. 2007.
[14] P. Heysters and G. Smit, *Mapping of DSP algorithms on the MONTIUM architecture*, Parallel and Distributed Processing Symposium, April 2003.
[15] A system description language, `http://www.systemc.org`
[16] A. Rose, S. Swan, J. Pierce and J. M. Fernandez, *Transaction level modeling in SystemC*, Open SystemC Initiative 2005.
[17] C. Ostler and K.S. Chatha, *An ILP Formulation for System-Level Application Mapping on Network Processor Architectures*, Design, Automation & Test in Europe (DATE '07), pp. 1–6, April 2007.
[18] D. Pham, S. Asano, M. Bolliger, M. N. Day, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa, *The design and implementation of a first-generation CELL processor*, IEEE International Solid-State Circuits Conference (ISSCC 2005), pp. 184–592, Vol.1, 2005.
[19] M. Butts, A. M. Jones, and P. Wasson, *A Structural Object Programming Model, Architecture, Chip and Tools for Reconfigurable Computing*, 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2007), pp. 55–64, 2007.
[20] Kazuyuki Hirata and John Goodacre, *ARM MPCore The streamlined and scalable ARM11 processor core*, Asia and South Pacific Design Automation Conference (ASP-DAC '07), pp. 747–748, 2007.
[21] *Seaforth-24 Device Data Sheet*, intellaSys, `http://www.intellasys.net`
[22] V. Zyuban and P. Kogge, *The energy complexity of register files*, the 1998 international symposium on Low power electronics and design, pp. 305–310, 1998.
[23] V. V. Zyuban and P. M. Kogge, *Inherently lower-power highperformance superscalar architectures*, IEEE Transactions on Computers, 50(3): 268–285, March 2001.
[24] Jos, C. Lorenzo, G. Antonio, lez, V. Mateo, and P. T. Nigel, *Multiple-banked register file architectures*, the 27th annual international symposium on Computer architecture, pp. 316–325, 2000.
[25] I. Park, M. D. Powell, and T. N. Vijaykumar, *Reducing register ports for higher speed and lower energy*, 35th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-35), pp. 171–182, 2002.
[26] André Seznec, Eric Toullec and Olivier Rochecouste, *Register Write Specialization Register Read Specialization: A Path to Complexity-Effective Wide-Issue Superscalar Processors*, 35th Annual IEEE/ACM International Symposium on Microarchitecture, (MICRO-35), pp. 383–394, 2002.
[27] W. Han, Y. Yi, M. Muir, I. Nousias, T. Arslan, and A. T. Erdogan, *Efficient Implementation of Wireless Applications on Multi-core Platforms based on Dynamically Reconfigurable Processors*, 2008 International Workshop on Multi-Core Computing Systems (MuCoCoS'08), 2008.
[28] M. Manjikian, *Multiprocessor enhancements of the SimpleScalar tool set*, Computer Architecture News (CAN), Vol. 29, Issue 1, pp. 8–5, March 2001.
[29] C. J. Hughes, V. S. Pai, P. Ranganathan and S. V. Adve, *Rsim: simulating shared-memory multiprocessors with ILP processors*, IEEE Trans. on Computer, vol. 35, Issue 2, pp. 40–49, Feb. 2002.
[30] M. M.K. Martin, et al., *Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset*, Computer Architecture News, Vol.33, No.4, pp. 92-99 September 2005.
[31] J. D. Davis, C. Fu and J. Laudon, *The RASE (Rapid, Accurate Simulation Environment) for chip multiprocessors*, ACM SIGARCH Computer Architecture News, Vol. 33 Issue 4, pp. 14–23, November 2005.
[32] P. M. Ortego and P. Sack, *SESC: SuperESCalar Simulator*, `http://iacoma.cs.uiuc.edu/~paulsack/sescdoc/` Dec. 2004.

[33] L. Benini, D. Bertozzi, A. Bogliolo, F. Menichelli, and M. Olivieri, *MPARM: Exploring the Multi-Processor SoC Design Space with SystemC*, Journal of VLSI Signal Processing, Vol. 41, pp. 169–182, Sept. 2005.

Edited by: Sabri Pllana, Siegfried Benkner
Received: June 16, 2008
Accepted: July 28, 2008

Scalable Computing: Practice and Experience
Volume 9, Number 3, pp. 197–206. http://www.scpe.org

ISSN 1895-1767
© 2008 SCPE

LATENCY IMPACT ON SPIN-LOCK ALGORITHMS FOR MODERN SHARED MEMORY MULTIPROCESSORS

JAN CHRISTIAN MEYER* AND ANNE C. ELSTER*

Abstract. In 2006, John Mellor-Crummey and Michael Scott received the Dijkstra Prize in Distributed Computing. This prize was for their 1991 paper on algorithms for scalable synchronization on shared memory multiprocessors, which included a novel spin-lock algorithm (a.k.a. MCS spin-lock). Their spin-lock algorithm distributes spin locations in memory to lessen the impact of bandwidth limitations. Their empirical work and architectural suggestions have since had a major impact on how the field has viewed spin-locks. Motivated by emerging architectures with an increasing number of cores, we present an empirical study on recent shared memory architectures, including IBM P5+ and SGI ccNUMA systems. Our results show that latency will have a much greater impact on performance than bandwidth on these and future architectures. Several testcases and a tabular overview of our results are included.

1. Introduction. Shared memory distributed computing is becoming more and more important as multi-core architectures are used to overcome the power and frequency "walls." With increasingly many cores fighting for shared resources, how resource allocation is handled will have a major impact on performance.

A process may request exclusive access to a resource by using a spin-lock which polls a shared data structure, "busy-waiting" or "spinning" until the state of the shared structure indicates that the lock is acquired by the requesting process. Spin-locks are typically used to protect short critical sections where the time to suspend/resume a requesting process is greater than the time spent on polling the shared variable. The design of a suitable data structure for spin-locks requires consideration of both scalable performance and fairness. As pointed out by Mellor-Crummey and Scott [1], interconnects may be saturated by a large number of requests for a shared data structure, and a poorly designed lock may cause starvation. This is further complicated by the multiple hierarchical memory levels of today's multi-core processors.

This study arose from looking at the cache coherency protocols of the Stanford Dash [8] architecture as adopted on SGI platforms with ccNUMA interconnects [7], and the outline of the Power5 cache system given by Kumar et. al. [9]. Both designs share the property that write requests are buffered enroute to their destination. This provides an opportunity to arbitrate memory access by transmitting negative acknowledgements to requests in transit before they are able to saturate the interconnect. Since this potentially alters the relationship between the actual performance of a shared spin location and the common preconception of their inefficiency, our study represents an attempt to quantify the significance of applying spin-lock algorithms which use a software approach to co-locate spin locations with their respective processors.

2. Historical Notes and Related Work. The use of a spin-lock algorithm for mutual exclusion was first suggested in a concise paper by Dijkstra [2], which proposed an algorithm which resolved conflicts due to concurrency by awarding the lock to the process which issued the last executed write request.

The acclaimed empirical work of Mellor-Crummey and Scott [1] from 1991 examined a range of spin-lock and barrier algorithms based on fetch-and-ϕ operations (a class of atomic read-modify-write instructions). Their work presented strong empirical evidence that significant performance improvements could be made by paying attention to how spin locations are distributed in memory. It also introduced novel algorithms for both barriers and spin-locks, most notably the aforementioned *MCS* spin-lock which exploits this bandwidth limiting effect. Part of their contribution was to show that the problems associated with contention for a shared location could be reduced by a sufficiently sophisticated software algorithm, although this was commonly thought to require special-purpose hardware at the time. Their work and architectural suggestions have had a major impact on how the field has since viewed the importance of local spins. Consequently, in a literature which spans the range from theoretical correctness proofs [6] through detailed program analysis and simulation [3, 4] to empirical studies [5], great attention has been given to ensure that locking algorithms spin only on memory addresses which can be co-located with the spinning processor, either in a local share of global memory, or in cache memory.

Magnusson et. al. [3] compare algorithms purely on the basis of the number of global memory references in their run-time analyses. Their paper includes detailed run-time analysis of three queue-based locks including the MCS lock, and propose two new locks which reduce lock release costs.

*Norwegian University of Science and Technology, Department of Computer and Information Science, Sem Sælands v. 7–9, NO-7491 Trondheim, Norway, {janchris, elster@idi.ntnu.no}

Michael and Scott [4] compare the performances of various implementations of fetch-and-ϕ operations, compare-and-swap and load-linked/store-conditional (LL/CS) in a simulated MIPS R4000 64-node multiprocessor, and make architectural suggestions for future multiprocessors based on their results.

A recent paper by Anderson and Kim [6] use a time complexity measure based on the number of remote memory references, which is defined as the number of references which cause interactions over the interconnect. They generalize spin-locking algorithms with respect to the atomic operation employed, and introduce a ranking system for the relative power of various fetch-and-ϕ operations. Given atomic operations of a suitable rank, they use the ranking system to prove time bounds for correct and starvation-free mutual exclusion on cache-coherent and distributed shared memory architectures.

Both Anderson and Kim [6] and Yang and Anderson [5] present asymptotic time complexities measuring time in terms of number of remote memory references.

3. Background. Given the amount of attention devoted to the concept of local spins in the literature published since the Mellor-Crummey and Scott article [1] established their significance, it is interesting to note that already at that time, cache coherent memory was observed to counteract the effect of contention in barrier algorithms. This is because the spinning associated with such algorithms mostly consists of extended strings of *read* operations in a wakeup phase, permitting a shared spin location to be cached (offloading the interconnect), and using the eventual cache invalidation as a broadcast wakeup signal. Note that simple spin-lock algorithms do not enjoy this property, since their spins consist of lots of *write* requests that require testing against a shared memory location, introducing traffic which may saturate the interconnect.

Many developments pertaining to the implementation of coherent cache memory have taken place since the saturation effect was first pointed out. In particular, current architectures come with notions of a memory hierarchy far more complex than the coherent cache/shared bus combination of the Sequent Symmetry which was used to establish the barrier results reported by Mellor-Crummey and Scott [1].

Based on these results, Michael and Scott [4] recommended using LL/CS to create atomic primitives for future architectures. Such instructions are hence included on modern shared-memory architectures such as the MIPS 14000 and IBM Power5. Our work empirically tests all of the spin-lock algorithms tested by Mellor-Crummey and Scott [1] on recent two recent larger shared-memory systems.

Our results question the conventional cost models which limits itself to local and remote accesses. Today's shared memory systems typically have several levels of memory hierarchy and is shown to be much more sensitive to latency rather than purely bandwith bound, as we will show in Section 6.

4. Target Platforms. Motivated by emerging multicore architectures with an increasing number of cores, we selected the following two platforms, both with 16 or more processors in a shared memory system as our test beds: an SGI Origin 3800 (up to 32 processors tested), and an IBM System p575+ (up to 8 dual cores tested). While these systems are not highly parallel in the multi-core sense, the authors believe that the communication issues faced by the larger scale interconnects of contemporary supercomputers will soon become relevant for emerging multi-core architectures. These architectures are rapidly approaching the same degree of parallelism, and are likely to confront the same "memory wall" which has precipitated the hierarchical memory structure of our test platforms, albeit on a chip-level scale.

The SGI Origin 3800 consists of "bricks" of 4 MIPS 14000 processors clocked at 600 MHz, which share a memory module via a crossbar. These 4-way units are interconnected in a fat tree. The system features a ccNUMA interconnect, which provides programs with a shared address space potentially spanning several physical racks of compute nodes. Transmissions of data across the interconnect is transparently handled via address translation, so that the system may be programmed as a symmetric multiprocessor. Memory traffic is handled by the same routing system regardless of the proximity of the communicating nodes, to provide the best approximation to uniform memory access the system can offer. Cache coherency in ccNUMA systems is maintained using a variant of the protocol in the Stanford Dash multiprocessor [8], using a directory approach where a snooping bus connects a small set of processors with their local part of the distributed memory. This bus is also connected with the local share of the directory, which maintains the caching status of the local lines, and forms an interface to the other parts of the directory [7].

The IBM System p575+ restricts shared memory to a node, which consists of 8 dual-core Power 5 processors, clocked at 1.9GHz. Each pair of cores share a level 2 cache, and 4 pairs are completely interconnected in a multi-chip module.

Detailed information regarding IBM System p575+ coherency protocol is difficult to find, but some information can be gleaned from Kumar et. al. [9], who give an explicit description of Shared Bus Fabric (SBF) design, stating that "Details of the modeled design [...] are based heavily on the shared bus fabric in the Power 5 multi-core architecture" [9]. Bearing in mind that the source is not an official technical document, it will still be used here as a high-level description of the Power5 coherence mechanism. Cache coherency is maintained using a "[...]MESI-like snoopy write-invalidate protocol[...]" [9], which is implemented with a set of unidirectional, pipelined buses and a hierarchy of dedicated units which arbitrate bus usage and queue requests.

Both platforms feature 'load linked' and 'store conditional' (LL/SC) instructions. These instructions work in pairs: LL is a read operation which marks a location as read, and a corresponding SC is allowed to write to the same location only if the value went unmodified in the meantime. This enables a processor to support the full range of fetch-and-ϕ operations without extending the instruction set, since they can be implemented in terms of short sequences of instructions with the final SC instruction failing if the entire operation was not carried out atomically.

We expect that the behaviors observed on these systems will be similar to future multicore architectures.

Although LL/SC can be used for all the semantic properties of fetch_and_ϕ operations, the SGI Origin 3800 also supports some fetch-and-ϕ operations natively, most notably fetch_and_increment, and fetch_and_decrement. Following the recommendations made by Michael and Scott [4], these depend on the use of special-purpose uncached memory (called *atomic reservoir memory* in the SGI Origin series). A similar mechanism is mentioned in the description of the Stanford Dash [8].

The properties of this feature are not examined here since atomic reservoir memory is not cached, making it less relevant with respect to examining effects of memory hierarchy on spin-locks.

5. Experimental Methodology. The empirical results in this study are divided into a set of general experiments which form an overview of lock performances on both target architectures, and a more specific set which validates the implications of the general experiments by isolating predictable effects. This reflects the timeline of the underlying research effort, as the general experiments form a preliminary evaluation of relative performances and tentative explanations, while later experiments address some of the questions raised in the initial phase.

Although the result material is reiterated below for the benefit of the discussion, the interested reader may trace this development, as the general experiments alone formed the basis of our previous conference paper [10].

The implementations used in the general experiments are based on pseudocode given by Mellor-Crummey and Scott [1]. The test suite includes implementations of a plain *test_and_set* lock, a version with exponential backoff, a *test_and_test_and_set* lock, the ticket lock with proportional backoff, as well as Anderson's lock and the MCS lock.

Modified versions of the *test_and_test_and_set* and *ticket* locks are tested on the IBM System p575+ only. The extensions made to these locks primarily address properties of the experimental setting, and are not intended as generally useful lock designs; their purpose is to further isolate the communication properties of locks which are prone to starvation.

In the interest of producing comparable results, the same program code was used in the general experiments on both platforms, except for a few conditionally compiled macros to handle the minor idiosyncrasies of each platform. All locks were implemented using *compare_and_swap* operations available in system libraries on the respective systems. Since the IBM System p575+ *compare_and_swap* does not preserve the comparison value when the operation fails, we wrapped the call in a conditionally compiled macro, in order to provide identical semantics on both platforms without introducing the extra overhead of a function call. We verified that the generated assembly code of our lock implementations in fact used the respective LL/SC instructions.

Since neither of the target platforms provides directly addressable local memory per processor, the effect of locality had to be reproduced by manipulating the memory layout of the lock data structures to exploit cache memory. This was done by padding the data structures so that each value which should be locally accessible was separated by the length of a cache line. As a control experiment, a modified version of the *Anderson* lock which was *not* padded was tested to see if it would result in a lock which consistently performed considerably worse than the original, padded *Anderson* lock. This predicted behavior is readily observable in the collected timings.

Each of the presented timing results represents the average lock acquisition time of 5000 locks, acquired and released in a tight loop, with or without a small critical section between acquisition and release. To reduce system induced variances, each presented result is the median of 75 runs.

Experiments with the modified *test_and_test_and_set* lock also monitored the fairness of tested algorithms, using the assumption that a fair lock will distribute the locks in all tests uniformly, awarding $\frac{N}{p}$ locks to each processor. The fairness results given are the median of 75 tests each reporting the maximal deviation from this expected value in a 5000 lock run.

5.1. Tested Locks. This section briefly describes each of the tested algorithms. We include all the locks described by Mellor-Crummey and Scott[1], as well as two versions we modified to isolate specific communication properties.

The *test_and_set* lock maintains a single global variable for locking, and has each contesting processor waiting in a tight loop, attempting to set it atomically. The *test_and_test_and_set* lock lowers the bandwidth requirements of the *test_and_set* lock by reading the present lock value to determine whether it can be acquired before attempting an atomic update. The *test_and_set* lock with exponential backoff, on the other hand, responds to a failed atomic update by waiting for a basic time period before attempting another update. This waiting period is doubled with each successive failure.

The modified *test_and_test_and_set* lock exploits the fact that the original lock reserves an entire memory location to store one out of two states. Using integers smaller than 0 to represent the open state, and positive integers to lock, it is possible to retain memory of which contestant won the last lock without significantly altering the amount of interconnect traffic. This is done by using an identifier for the acquiring process(or) as the lock value, and releasing it by negating this number. It is thereby possible to prevent the lock from being acquired twice by the same contestant, at the cost of a single local comparison operation.

The *ticket* lock maintains two global counters which track acquisition attempts and lock releases separately, effectively forming a FIFO queue (thus eliminating starvation). Each failed attempt to acquire the lock results in a waiting period which is proportional to the length of the queue at the time of the attempted acquisition.

The modified *ticket* lock uses the same mechanism, but only forces an adjustable-length tail of the queue to back off and wait. The remaining pool of processors at the head of the queue participate in a contest for a *test_and_test_and_set* lock. This modification makes the lock prone to starvation, as a single processor can enter the pool and be bypassed any number of times. The purpose of testing this lock is that the permitted length of the queue effectively puts a limit on how many lock acquisitions must take place between successive acquisitions by a single processor. It should be noted that this technique subsumes the modified *test_and_test_and_set* lock as a special case (length 1). The extra overhead of managing two structures, however, incurs an overhead which means that it is not competitive with other tested locks; results are included mainly to provide a test environment for comparing the results of variable queue length.

The *Anderson* lock uses a queue like the *ticket* lock, but distributes the target locations of the waiting spins in an array. The array locations are chosen so that only the spinning processor and its predecessor in the queue are accessing them. This attempts to reduce contention to a greater extent than the global counters of the *ticket* lock, while preserving its starvation freedom.

The *MCS* lock uses a similar construct, but replaces the array in the *Anderson* lock with a linked list, moving the significant part of the lock data structure into processor-local memory. When implemented using coherent caches as processor-local memory, its main difference from the *Anderson* lock is that the *MCS* lock requires each contesting processor to enter the linked list by performing a remote write operation to its predecessor in the list, to communicate its own spin location.

6. General Experiments. The results of the general experiments are presented in Figures 6.1 through 6.5.

Fig. 6.3 presents the same material as Fig. 6.2, except for the removal of the *test_and_set* lock with exponential backoff, to emphasize the differences between the remaining locks using a more appropriate scale.

Tables 6.1, 6.2 summarize the result material for the tests with a critical section, describing relative lock performance and scalability for each combination of target architecture and lock type. The results with immediate lock release are omitted from these summaries for the sake of brevity, since the tests with critical section both capture all behavioral differences, and are likely to be of greater practical importance. Acquisition time is categorized according to performance relative to other locks tested under the same conditions. Scaling properties are noted where a clear tendency is visible already by the limited number of processors in the results. Locks which exhibit favorable scaling properties are labelled with the highest number of processors for which this is observed. Entries of particular significance to our conclusions are highlighted in boldface.

The most striking feature of the presented results can be found in the difference between Figures 6.1 and 6.2, where the *test_and_set* lock with exponential backoff goes from displaying favorable acquisition times with no

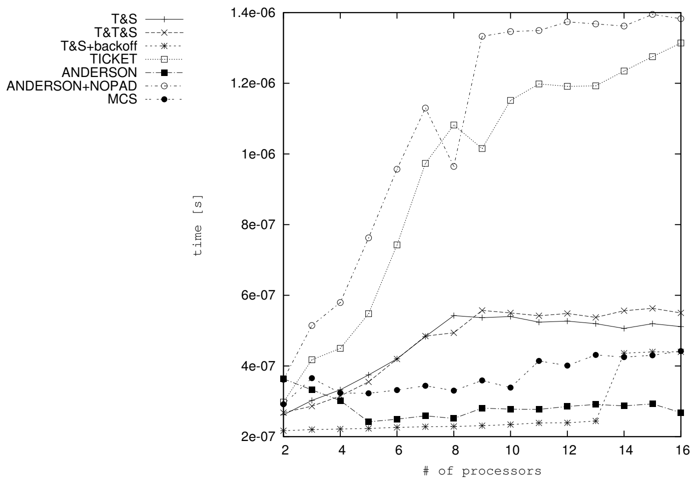


Fig. 6.1. *Performance of all locks on IBM System p575+, no critical section.*

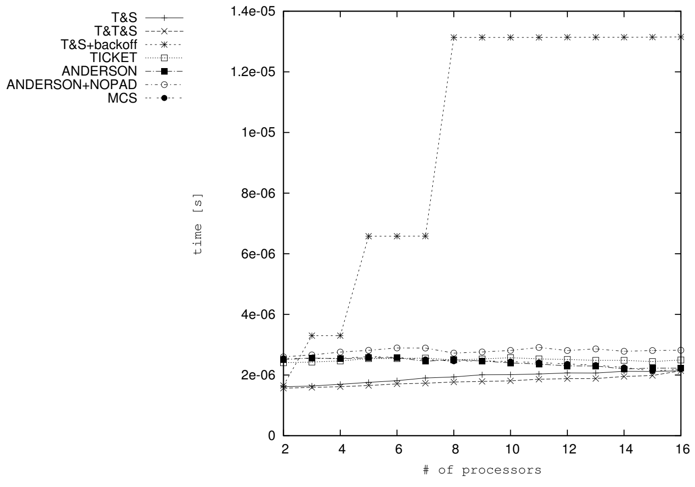


Fig. 6.2. *Performance of all locks on IBM System p575+, small critical section.*

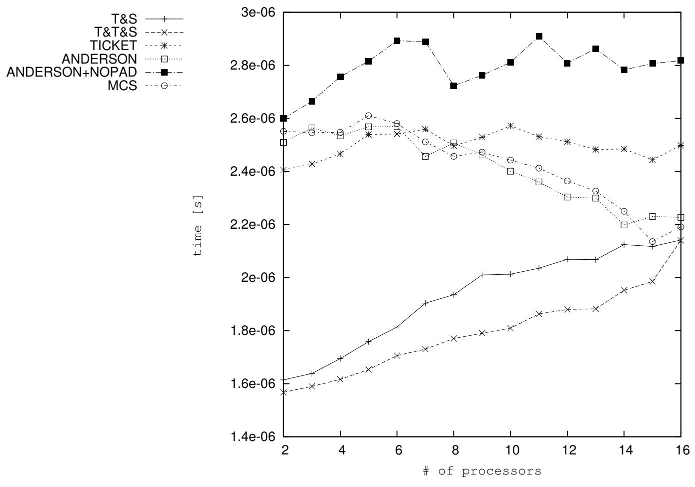


Fig. 6.3. *Performance of selected locks on IBM System p575+, small critical section.*

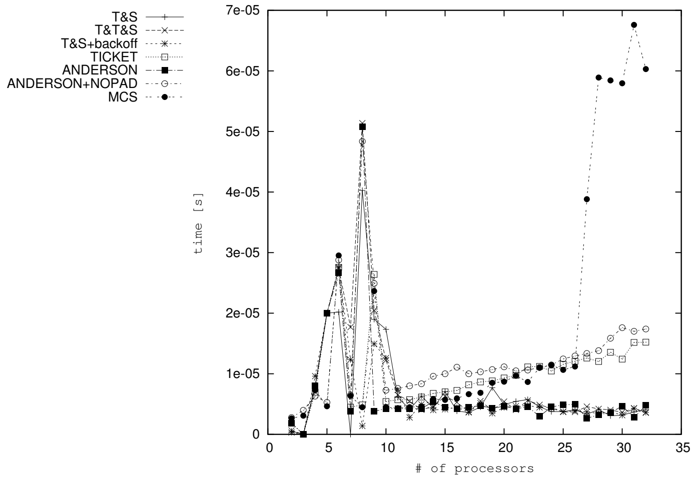


Fig. 6.4. *Performance of all locks on SGI Origin 3800, no critical section.*

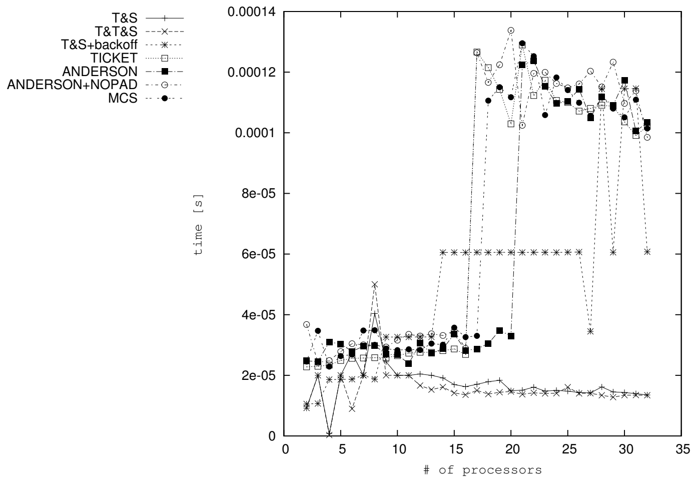


Fig. 6.5. *Performance of all locks on SGI Origin 3800, small critical section.*

critical section, to showing drastic increases when there is a critical section. In Fig. 6.2, the exponential backoff is visible in the shape of the graph. It is evident that the waiting periods dominate lock acquisition time, as a given number of processors implies that acquisition time becomes proportional to one of the power-of-two multiples of the basic delay. The slope of this effect will depend on that basic waiting time as well as the degree of contention, but the characteristic behavior indicates that this lock needs to be tuned with a number of processors in mind, and thus is poorly scalable. The timings from the SGI Origin 3800 in Fig. 6.4, 6.5 display the same behavior, although slightly more erratically.

Note that Fig. 6.5 in particular displays a sharp jump in acquisition times, corresponding with the need to traverse another level of switching for address translation to complete the memory access requests of the starvation-free locks. This consideration leads us to the central observation of this work, which is that when considering the performance implications of local spins, *interconnect latency has supplanted limited bandwidth as the primary reason for degradation*, at least on the architectures under examination. The presented results are evidence of this to this in two ways:

1. The variations on the *test_and_set* lock display scalable characteristics in all the general tests. This would not be true if acquisition time was dominated by the hardware struggling to serialize an increasing number of write requests for a single, shared location.
2. The performance of the *ticket* lock appears similar to the *MCS* and *Anderson* locks, particularly in the cases where a critical section is present. The common property shared by these locks is that they

TABLE 6.1
Summary of results with critical section, IBM System p575+

Lock type	FIFO	Acq. time	Scaling
Test&set *Test&test&set*	No	**Low**	#cpus bound
Test&set *w.backoff*	No	High	#cpus bound
Ticket *Anderson* *MCS*	Yes	High	Scales for $p < 16$

TABLE 6.2
Summary of results with critical section, SGI Origin 3800

Lock type	FIFO	Acq. time	Scaling
Test&set *Test&test&set*	No	**Low**	Scales for $p < 32$
Test&set *w.backoff*	No	High	#cpus bound
Ticket *Anderson* *MCS*	Yes	High	**Latency bound**

maintain a FIFO ordering on lock acquisitions, forcing a modest number of remote write operations when the lock is handed to a remote process. The time taken to perform these remote writes visibly dominate the performance degradation due to contention for a shared location: on the IBM PSeries 575+ there is a small benefit from laying out spin locations in processor-local caches (Fig. 6.3), while the SGI Origin 3800 shows no observable advantage (Fig. 6.5).

The implication for lock design is that the benefits from the effort of ensuring that spin locations are tightly bound to each single processor may reasonably be called into question. Instead of asking how many remote write operations are required for a given locking algorithm, the presented results indicate that it is equally relevant to ask exactly how remote these operations are likely to be.

The results for less than 10 processors on the SGI Origin 3800 are inconclusive with respect to which lock has the superior acquisition time, particularly for the cases with immediate release in Fig. 6.4, but also for the *test_and_set* and *test_and_test_and_set* locks with critical sections in Fig. 6.5. A relevant comment to this is that during the experiments, it proved particularly tricky to get any measure of stability from timings collected on this machine. This is at least partly due to the fact that the transparent address translation of the ccNUMA architecture gives the operating system great freedom in the (re-)scheduling of processes at run time, effectively making it extremely difficult to predict the layout of processes. The noisy result material is included here for completeness, modest as its information value may be for comparisons.

7. Experiments with the Modified Test_and_Test_and_Set Lock. Figure 7.1 plots acquisition times in a repetition of the critical section experiment on IBM System p575+, with the addition of the modified *test_and_test_and_set* lock.

The two simple *test_and_set*-type locks appear to be superior to all other tested algorithms in Figures 6.3, 6.5. The common trait shared by these locks is that they contain the possibility of starvation. The observation that communication latency dominates lock acquisition times predicts low acquisition times for these locks when acquiring multiple locks in a tight loop, as a processor which has already cached the lock structure will have a significant advantage over others in the next acquistion.

The modified *test_and_test_and_set* lock is designed to eliminate the possibility for a single processor to monopolize the lock, effectively enforcing that each acquisition will pass the lock structure far enough down the memory hierarchy to reach a level accessible by multiple cores. It is still prone to starvation by permitting a small set of processors to pass the lock between them.

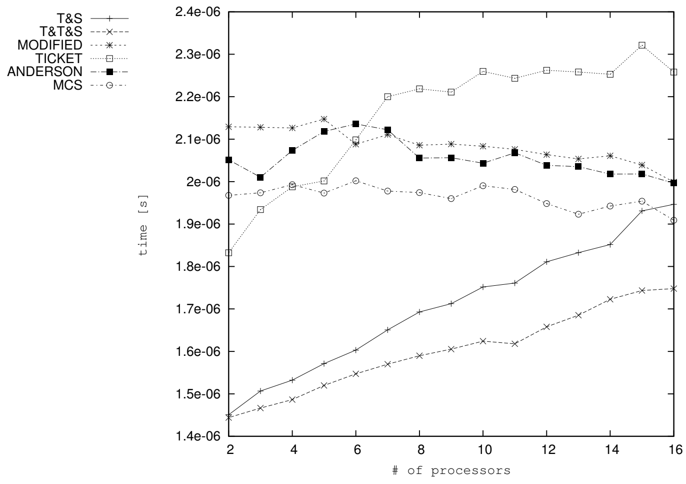


FIG. 7.1. *Performance with modified Test_and_Test_and_Set lock and critical section*

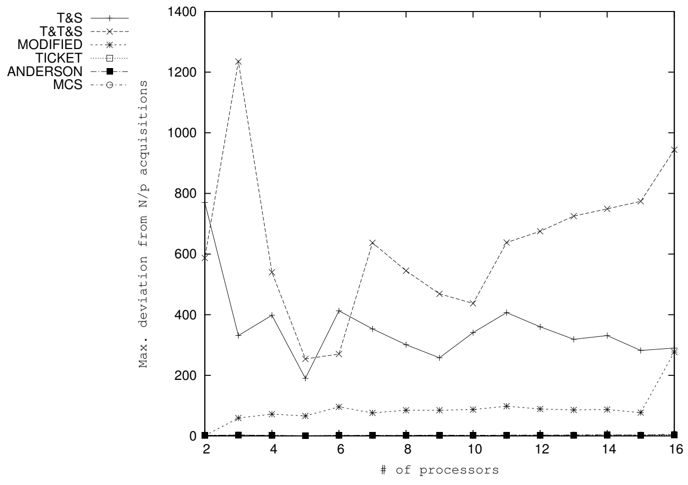


FIG. 7.2. *Fairness with modified Test_and_Test_and_Set lock and critical section*

The impact of this is observable in Figure 7.2, which plots deviations from the assumption that a fair algorithm will distribute locks uniformly among participating processors. The validity of this measure of fairness is supported by the fact that all the starvation-free locks display near perfect fairness. It is evident that the observed run times from both of the *test_and_set*-type locks are heavily biased by the increased speed of a single processor re-aquiring a recently released lock, given the marked improvement in fairness which stems from explicitly disallowing this behavior.

The other interesting feature of this experiment is that the resulting acquisition times for the modified lock closely follow those of the *Anderson* lock in Figure 7.1. Both lock structures force a remote access for each acquisition and release. This agrees with the observation that increased contention for a single memory location has an insignificant effect compared to the overhead of accessing any remote memory. Predicting acquisition time is nevertheless more complicated than counting the number of remote accesses. This is shown by the *MCS* lock outperforming both the modified *test_and_test_and_set* and *Anderson* locks in spite of requiring at least as many remote accesses for a typical acquire/release cycle, and more in the worst case.

Given the otherwise similar structure of the *MCS* and *Anderson* locks when implemented using cache coherency and padding, the main difference which separates their operation is that the *MCS* lock requires each new member of the ordered queue to register by its predecessor. This invalidates the cache line which holds its spin location, while the *Anderson* lock only invalidates the spin locations of successors, waking them when they acquire the lock. When a successor in the *MCS* algorithm causes the lock of its predecessor to be re-fetched

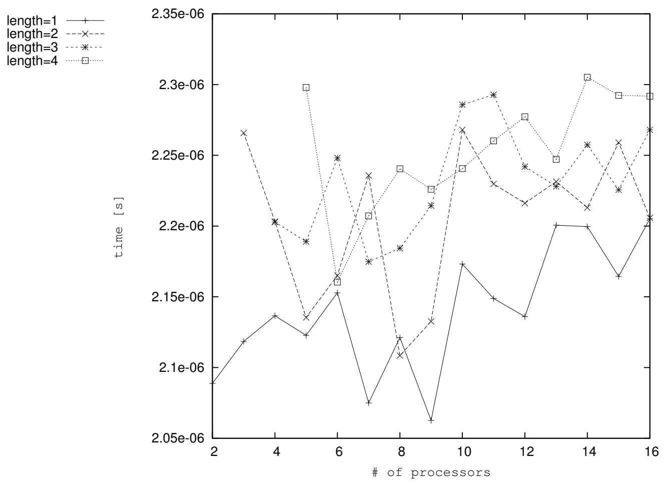


FIG. 8.1. *Comparison of modified ticket locks with variable queue length*

while it is being released, the resulting memory transfer will have some of its latency masked as a side-effect of the former invalidation. This predicts that the *MCS* lock's advantage is related to the fact that a short critical section may release the lock before a line has formed, while a long critical section will complete the establishment of a longer queue before the lock is released to its head.

Figure 6.1 shows that the absence of any critical section gives an advantage to the *Anderson* lock. We found that repeated testing involving only the *Anderson* and *MCS* locks shows that their run-time characteristics become indistinguishable when adjusting the length of the critical section by factors $\frac{1}{5}$ and 10, but distinguishable between these points. This is a curious effect to observe on an architecture which otherwise should suit the cost model of local vs. remote access (featuring private L2 caches and a crossbar interconnect [9]). As consistent performance advantages are still tied to reduced latency, and increasingly nonuniform interconnect performance can be expected from future architectures, this further supports the argument that practical communication cost models must account for processor locality to enable the differentiation of faster and slower remote accesses.

8. Experiments with the Modified Ticket Lock. The design of the modified *ticket* lock is intended to elicit similar effects to the modified *test_and_test_and_set* lock, by forcing each successive lock acquisition by a single processor to be separated by an adjustable number of other acquistions. The reason for this experiment is that the 16 cores on the test system are distributed as 8 dual cores, and the shared bus fabric of the Power5 design of the IBM System p575+ implements its snoopy cache coherency protocol by a dedicated bus which shortcuts the interconnect to update cache lines between cores on the same chip. Such updates would give an advantage to neighboring cores when acquiring a lock which does not enforce ordering. The behavior of the modified lock can hence be expected to alter when the queue length exceeds the number of cores per chip, yielding an estimate of the magnitude of the advantage lost.

Figure 8.1 shows that for the IBM System p575+, the expected change in behavior is observable between queue lengths 1 and 2. While the modified lock design naturally prevents a full set of measurements to be obtained for $p < length$, there is still a clear indication that queue length 1 gives a measurable performance advantage, while no systematic differences were found for longer queues.

This observation adds weight to the argument that future communication cost models will need locality details. While the significance of the observable difference between closely and loosely coupled interconnect on the test platform is small, its significance can be expected to increase with growing numbers of cores on a chip.

9. Conclusions and Future Work. The larger scale interconnects of recent supercomputers such as IBM Power systems and SGI's Origin series may soon become relevant for large multi-core processor systems. This article hence examines the performance of several spin-lock algorithms on such systems.

In contrast to observations from older architectures [1], this work shows that neither the *test_and_set* nor the *test_and_test_and_set* lock suffers degraded acquisition times with upscaling. Modern interconnects are not saturated by the bandwidth requirements of the tested algorithms. Unlike the older test systems which had severe bandwidth restrictions, modern interconnects feature highly connected topologies such as crossbars and fat trees. Increasing the connectivity of the interconnect greatly improves the aggregate bandwidth of the system, but has a more moderate impact on latency. Modern systems also feature snoopy cache coherency protocols between on-chip cores. Our experiments showed that locks performing well on such systems favored processes on processors near the one which last held the lock. Unfortunately, this impacts fairness.

Secondarily, the scaling properties of the *ticket* lock was observed to be similar to those of the *Anderson* and *MCS* locks. The assumption that the *ticket* lock's use of a shared spin location would saturate the interconnect and cause performance degradation, led to the prediction that the *ticket* lock should scale poorly compared to the *Anderson* and *MCS* locks. However, in the practically useful cases, where the lock protects a critical section, such an effect was visible to only a moderate extent on the IBM System p575+, and not at all on the SGI Origin 3800. Our conclusion is that the interconnect bandwidth of the test platforms is sufficient to reasonably handle the greater demands of the *ticket* lock, because of its similar performance to the other starvation-free locks.

Thirdly, the remote writes of both the *ticket*, *Anderson* and *MCS* locks visibly dominated acquisition time in the same way for all three locks when the distance to remote memory increased on the SGI Origin 3800. This showed that the latency of these remote writes had a more significant effect on acquisition time than saturation of the interconnect due to the greater amount of traffic generated by the *ticket* lock. This conclusion was supported by further experiments on the IBM System p575+. There it was found that starvation-prone locks achieve favorable run time when awarding locks to a single processor many times in succession. Forcing these locks to perform remote writes resulted in improved fairness, but also in run times comparable to those of the poorer performing starvation-free locks. This demonstrated that the same latency penalty applied when locks were transferred across the interconnect.

Finally, it was shown that lock exchanges between neighboring cores permitted a small performance improvement on the IBM System p575+.

Our tests only include dual-core processor nodes, but the lower latency exhibited between the cores indicates that models of future many-core architectures should distinguish between multiple levels of interconnect. Locking algorithms with a detailed measure of distances to remote memory is hence an interesting topic for further study.

Acknowledgements. The authors thank Prof. Lasse Natvig at NTNU and Helge Rustad at SINTEF for their valuable feedback on early versions of this work. Magnus Jahre and Rune E. Jensen at NTNU provided enlightening discussions and meticulous scrutiny of the experimental code. The many useful comments of the referees are also appreciated.

REFERENCES

[1] J. M. Mellor-Crummey and M. L. Scott, *Algorithms for Scalable Synchronization on Shared-Memory Architectures*, in ACM Transactions on Computer Systems, Vol. 9, No. 1, 1991, pp. 21–65.

[2] E. W. Dijkstra, *Solution of a problem in concurrent programming control*, in Communications of the ACM, Vol. 8, No. 9, 1965, pp. 569.

[3] P. Magnusson, A. Landin and E. Hagersten, *Queue Locks on Cache Coherent Multiprocessors*, in Proceedings of the Eighth International Parallel Processing Symposium, 1994, pp. 26–29

[4] M. M. Michael and M. L. Scott, *Scalability of Atomic Primitives on Distributed Shared Memory Multiprocessors*, University of Rochester Computer Science Department, Tech. rep. #528, July 1994.

[5] J-H. Yang and J. H. Anderson, *A Fast, Scalable Mutual Exclusion Algorithm*, in Distributed Computing, Vol. 9, No. 1, 1995, pp. 51–60.

[6] J. H. Anderson and Y-K. Kim, *A Generic Local-spin Fetch-and-ϕ-based Mutual Exclusion Algorithm*, in Journal of Parallel and Distributed Computing, Vol. 67, Issue 5, 2007, pp. 551–580.

[7] J. Laudon and D. Lenoski, *The SGI Origin: A ccNUMA Highly Scalable Server*, in ACM SIGARCH Computer Architecture News, Vol. 25, No. 2, 1997, pp. 241–251.

[8] D. E. Lenoski, J. Laudon, K. Gharachorloo, W-D. Weber, A. Gupta, J. Hennessy, M. Horowitz and M. S. Lam, *The Stanford Dash multiprocessor*, in IEEE Computer, Vol. 25, No. 3, 1992, pp. 63–79.

[9] R. Kumar, V. Zyuban and D. M. Tullsen, *Interconnections in Multi-core Architectures: Understanding Mechanisms, Overheads and Scaling*, in ACM SIGARCH Computer Architecture News, Vol. 33, No. 2, 2005, pp. 408–419.

[10] J. C. Meyer, A. C. Elster, *Latency Impact on Spin-Lock Algorithms for Modern Shared Memory Multiprocessors*, in Proceedings of CISIS2008, IEEE Computer Society CPS, March 2008, pp. 786–791.

Edited by: Sabri Pllana, Siegfried Benkner
Received: June 16, 2008
Accepted: July 28, 2008

Scalable Computing: Practice and Experience
Volume 9, Number 3, pp. 207–218. http://www.scpe.org

ISSN 1895-1767
© 2008 SCPE

PARALLEL ADVANCED VIDEO CODING: MOTION ESTIMATION ON MULTI-CORES

SVETISLAV MOMCILOVIC* AND LEONEL SOUSA*

Abstract. The new Advanced Video Coding (AVC) standards further exploit temporal correlation between images on a sequence by considering multiple reference frames and variable block sizes. It increases the compression rate for a given video quality at the cost of a significant increase in the computational load. Specialized hardware processors have been proposed to perform real time motion estimation on AVC, but the non-recurring engineering cost of these solutions is too high. This paper proposes a parallel algorithm that exploits the capacity of the current multi-core processors to implement real time motion estimation for AVC. In particular, exploiting the computational capacity and the fast memory system of the heterogeneous multi-core CELL processor, the synergetic processors accelerate the motion estimation while the main processor executes in parallel the other components of the AVC system. Experimental results show that motion estimation can be performed in less than 50ms per frame, for CIF video format, with up to 5 reference frames and variable block sizes, by programming the CELL with the proposed parallel algorithm. In addition, the scalability of the proposed solution is proven regarding the video sequence resolution, the number of cores and reference frames used.

Key words: motion estimation, AVC, CELL

1. Introduction. Motion Estimation allows the reduction of temporal redundancy in video sequences, but it is the most computationally expensive component of a video encoder. The mostly adopted Block Matching Motion Estimation (BMME) technique, divides each frame in rectangular blocks, called Macroblocks (MBs), which are basic Motion Estimation (ME) units. ME applies a search algorithm on the MBs to find the best match in a Reference Frame (RF), according to a distortion measure based on the Sum of Absolute Differences (SAD).

The newly introduced techniques proposed by the H.264/MPEG-4 AVC coding standards, such as Rate-Distortion Optimization (RDO), variable MB sizes and Multiple Reference Frame Motion Estimation (MRF-ME) [1], highly improve the coding quality. However, they drastically increase the involved computational load, being estimated that ME typically represents up to 80% of the whole set of computations, when MRF-ME and the optimal Full-Search Block-Matching (FSBM) [2] are used. With the high demands of the H.264/AVC video coding standard, it is very hard to implement motion estimation in real time, especially when a full configuration with 5 RFs and 7 different MB shapes is considered. To overcome this problem, adaptive search algorithms, such as Hybrid Unsymmetrical-cross Multi-Hexagon-grid Search (UMHexagonS) [3] have been adopted and dedicated processors have been developed [4]. Recently, multi-threading multimedia processors have been realized, namely for implementing an H.264 video coder based on a fast ME search algorithm [5]. The main thread running on the RISC processor is responsible for controls and synchronization of data communication among threads, while the subordinate threads, running on specialized cores, are used for multimedia acceleration, such as parallel SAD calculation. The current MB and Search Area (SA) are cached on shared on-chip memory.

Other approach is to program multimedia applications on heterogenous multi-cores, such as the Sony Computer Entertainment, Toshiba, and IBM (STI) Cell Broadband Engine Processor. The Cell heterogenous multi-core processor integrates two groups of cores [6]. One is a general purpose 64bit Power Processor Element (PPE), which contains a Power Processor Unit (PPU) with access up to 512MBytes of external memory; the other cores are Synergistic Processor Elements (SPEs), with specialized dual issue 128bit architecture, and a total of 256kBytes of Local Storage (LS). Each SPE is composed of two components: a Synergistic Processor Unit (SPU) and a Memory Flow Controller (MFC). Moreover, specialized instructions are offered in the SPU intrinsic library [7].

In [8], an H.264 video encoder is used as a part of a video surveillance system implemented on the STI Cell Broadband Engine Processor. The ME is implemented on a single SPE, only considering one RF and a fixed MB size. Frame data is partitioned and transferred to the SPE in a row basis. However, no information is provided about the adopted search algorithm.

In [9], a parallel H.264 video coder was implemented in the Cell. Video Coding (VC) components are implemented in different SPEs, and successive partitions of a frame are passed through the pipeline. A single frame is divided into four 16Byte aligned partitions. The PPE is responsible for data partitioning and scheduling. By using the SPU Single Instruction Multiple Data (SIMD) instructions the SAD is computed in parallel. Like

*INESC-ID/IST TULisbon, Rua Alves Redol 9, 1000-029, Lisboa – PORTUGAL,
{Svetislav.Momcilovic, Leonel.Sousa}@inesc-id.pt

in [8], the ME is performed just over a single RF, using a fixed MB size and a fast search algorithm. Experimental results, regarding encoding and ME time, are not provided.

In [10], a parallel ME algorithm for multi-core architectures is described and implemented in the Cell processor. A VC is running in the main processor and the ME is offloaded to the SPEs. This algorithm considers several RFs, and each RF is processed in a different SPE. The same Current Frame (CF) is transferred to each SPE and one row is processed in each iteration. However, this approach is not scalable, namely referring to the number of cores and RFs. Moreover, the presented time results just consider the time that VC has to wait for ME to be finished, and not the ME processing time in the SPEs.

The present work starts with the algorithm presented in [10], improves it to obtain a scalable solution that achieves efficient ME independently of the number of RFs and cores. This scalability, which is also extended to the image resolution is achieved in a row-based communication and computation approach. The presented time for ME on SPE can be hidden behind the rest of the VC executing in parallel in the PPE. Experimental results for High Definition (HD) video sequences are also presented.

The rest of the paper is organized as follows. In Section 2 it is presented the parallelization approach, which includes the new scalable solution. Section 3.1 describes the implementation of the improved algorithm on the Cell platform. Section 4 presents experimental results, including the ones for ME time on the SPE side, and also for the HD test video sequences. Finally, Section 5 concludes this paper.

2. Parallelization Approach. Although the proposed parallelization approach is general for multi-core processors, for simplicity it will be explained here referring to the CELL architecture.

2.1. Basic parallel algorithm. The Asymmetric-Thread Runtime programming model [11] is adopted in the proposed parallelization approach. The PPE runs the main task, which includes almost all video encoding process, except the ME part, and the control procedure. The ME, as the most computationally intensive part, is divided into threads running in the various SPEs. Each SPE runs a unique thread, performing Direct Memory Access (DMA) data loading from the main memory and executing the ME algorithm in an autonomous way.

Figure 2.1 shows the proposed parallelization approach. It exploits data parallelism by applying the same search algorithm to the different RFs on the various SPEs. The optimal configuration of the H.264/AVC video encoder in terms of quality versus bit rate uses 5 different RFs. Therefore, with this approach, up to 5 SPEs can be used if we want resources to be busy. Pixels from different RFs are loaded from the frame to the SPEs' LS, while pixels from the CF are loaded at the same time in the LS of all SPEs. When the ME process is finished, results are passed back to the PPE, that is responsible for the remaining components of the video coding process.

ME is running in threads in parallel with the main VC process. When results are received, ME for the next row can start and run in parallel with the rest of the VC, as it is shown in Figure 2.2.

An SPE has access to the data frame in chunks according to the MB organization. Each data chunk is composed by a row of MBs from the CF and a row of SAs from RF. The pixels in a data chunk are loaded into the LS of an SPE doing one DMA transaction. This "row oriented" approach is used to maximize data reuse: for the first row of MBs and SAs all pixels have to be loaded, but for the next ones only the pixels not present in the intersection of two consecutive rows of SAs have to be transferred. Figure 2.3 shows the initial data from both CF and RFs that need to be loaded, as well as the additional pixels that need to be loaded for the next two MB/SA rows in two individual DMA transactions. Figure 2.3 *b)* illustrates the SAs that have to be loaded in consecutive rows; only non-overlapping regions have to be transferred to the SPEs' LS.

In the rest of the paper, a MB is considered to have 16×16 adjacent pixels and all sub-units in a MB, independently of their size, will be called *sub-blocks*. The order by which the MB and sub-blocks are processed is decided by the SPEs. Process is repeated for each MB in a row. When all MB are processed, the calculated Motion Vector (MV)s and the distortion measures are passed back to the PPE in a single DMA transaction.

Synchronization points are introduced whenever DMA transfers for either pixels or results are finished, allowing data buffers reuse. It is worth to notice that, in the case of specialized processors, memory is a sparse resource; and it could be very useful to save memory for robust search algorithm implementation purposes.

The SPU and MFC allow independent execution of computation and data transferring in an SPE [12]. This feature is exploited in the proposed parallel programming approach by overlapping processing and memory accesses through the double buffering technique. Figure 2.4 shows the proposed parallelization approach, namely the progress in time of DMA transfers and data processing. The DMA transfers and data processing segments are presented with boxes, while the double lines represents synchronization points. The algorithm performs one

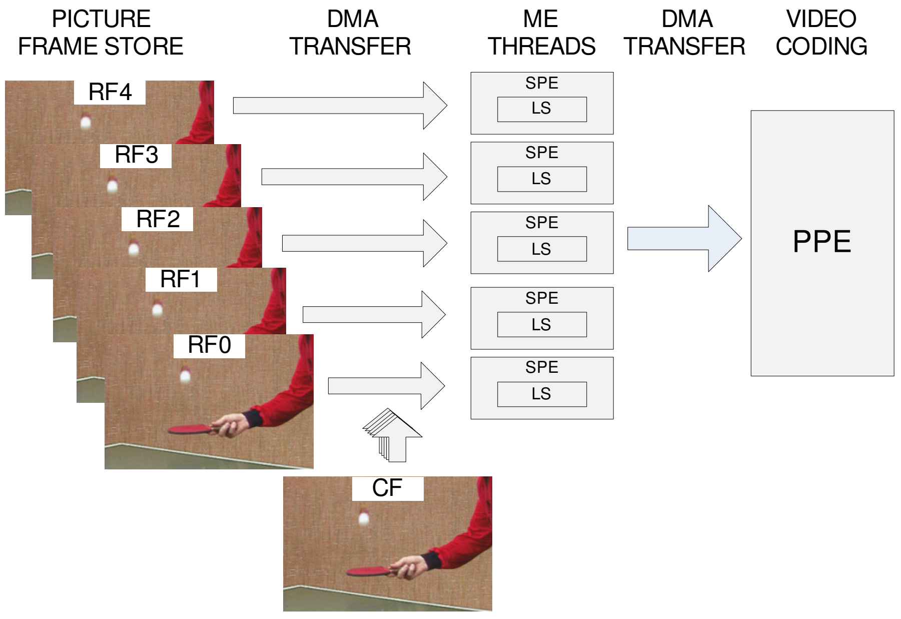


FIG. 2.1. *Parallelization approach.*

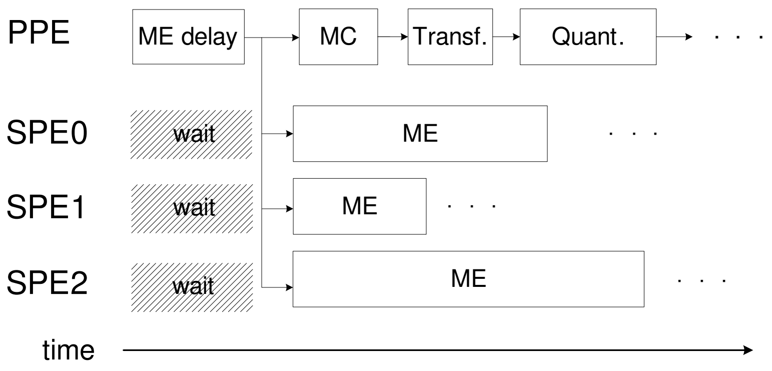


FIG. 2.2. *Time diagram of described VC approach.*

iteration for each MB row (n is the number of MB rows). Firstly, in *Iteration 0*, each SPE loads the initial row of MBs and the corresponding row of SAs in the *pels0* buffer. In the next iterations, in parallel with the processing of data in *pels(i mod 2)* buffer, data are loaded into the *pels((i+1) mod 2)* buffer. After each iteration exits a synchronization point is introduced, in order to ensure that data to be processed in next iterations is already loaded into the LS. It is important to mention that n does not represent the number of rows in a frame, but the total number of rows for the whole sequence, because the loading of the first row of the next frame can be performed in parallel with the processing of the last row of the current frame. In such a way the pipeline processing/loading is maintained, saving the time required to individually transfer the initial row.

Figure 2.5 presents in detail the steps of one iteration of the proposed algorithm both in the PPE and the SPE, where SPU and MFC are represented separately. When the PPE already prepared MB/SA buffers for row $k + 1$, a DMA transfer is started. Then, PPE waits for a signal from the SPE announcing that the ME results for the row $k - 1$ are ready. When the signal arrives, the PPE uses the available results to continue the VC of row $k - 1$. As soon as the reading of the buffer for row k is finished, SPU can start ME for row k. This ME process is executed in the SPU in parallel with the DMA transfer of row $k + 1$ performed by the MFC. When the ME is finished, results are sent to PPE through a new DMA transfer. As soon as the DMA transfer is finished, a signal is sent to the PPE, in order to continue VC.

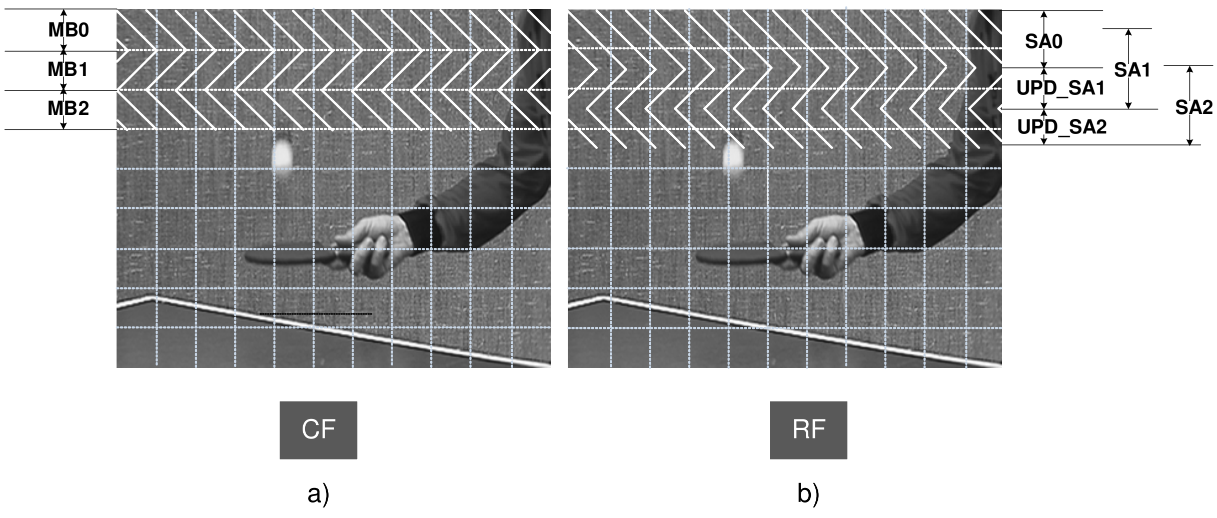


FIG. 2.3. *Frame division for data transfer: a) current frame b) reference frame.*

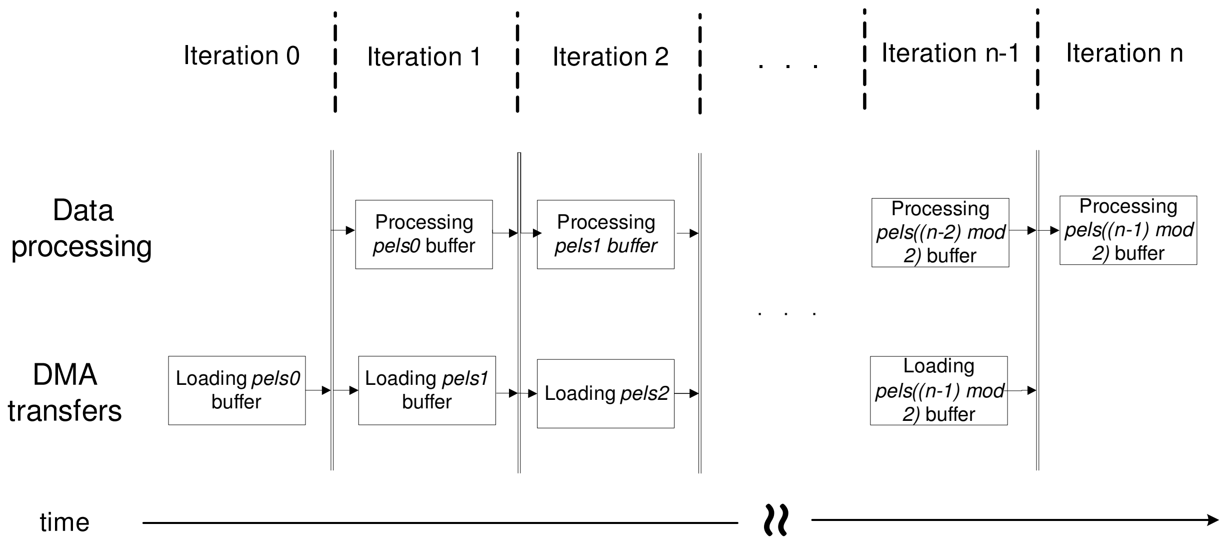


FIG. 2.4. *Double buffering in proposed algorithm.*

2.2. Scalable algorithm. The proposed basic parallel algorithm is efficient but it does not scale, in the sense that the maximal parallelism achieved, which means in this case the maximal number of used cores is limited by the number of RFs. In order to achieve a scalable algorithm, CF and related RFs can be divided in sub-CFs and sub-RFs, corresponding in this case to vertical slices. An example with 6 SPE cores and 2 RFs is presented in Figure 2.6. The scalable algorithm is obtained by applying the basic parallel algorithm to the sub-CFs and the sub-RFs, wich can be seen as full CF and RF. A slight overhead is introduced to organize the vertical slices, but the advantage is that the algorithm is able to use any number of cores to perform in parallel the ME independently of the number of RFs used. The same approach can be followed if the video sequence resolution is so high, that a single MB/SA row can not fit in the LS.

3. Motion Estimation.

3.1. An Implementation of the H.264/MPEG-4 AVC on the CELL processor. The proposed parallel algorithm is programmed in the JM 14.0 H.264/MPEG-4 AVC software [13]. Only P frames are considered in this work. Encoder uses the maximum number of 7 different sub-block types, multiple RFs and full pixel precision. Synchronization between the PPE process and the threads executing on the SPEs is

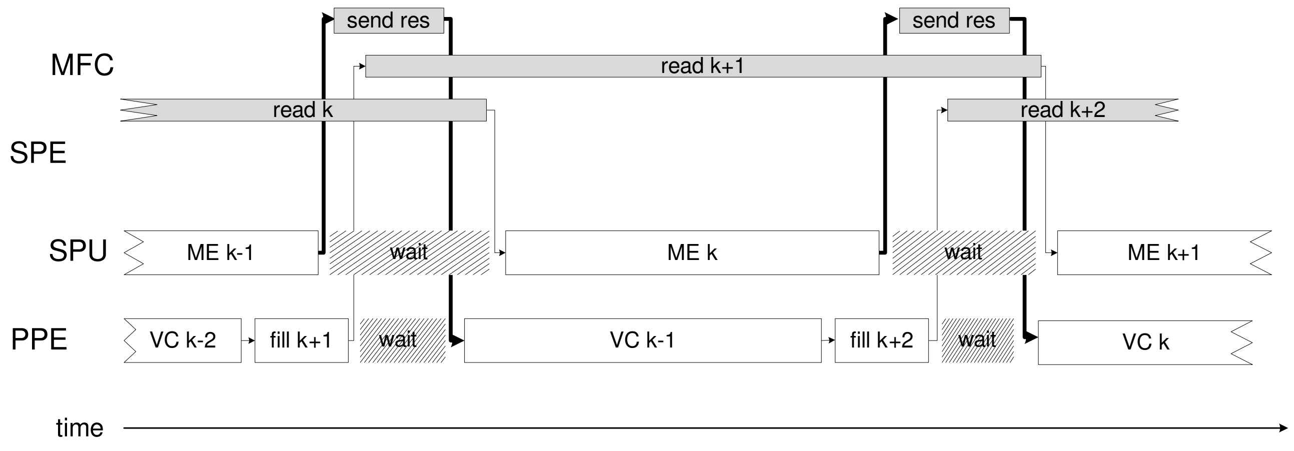


FIG. 2.5. *Data flow and phases in an iteration of proposed diagram.*

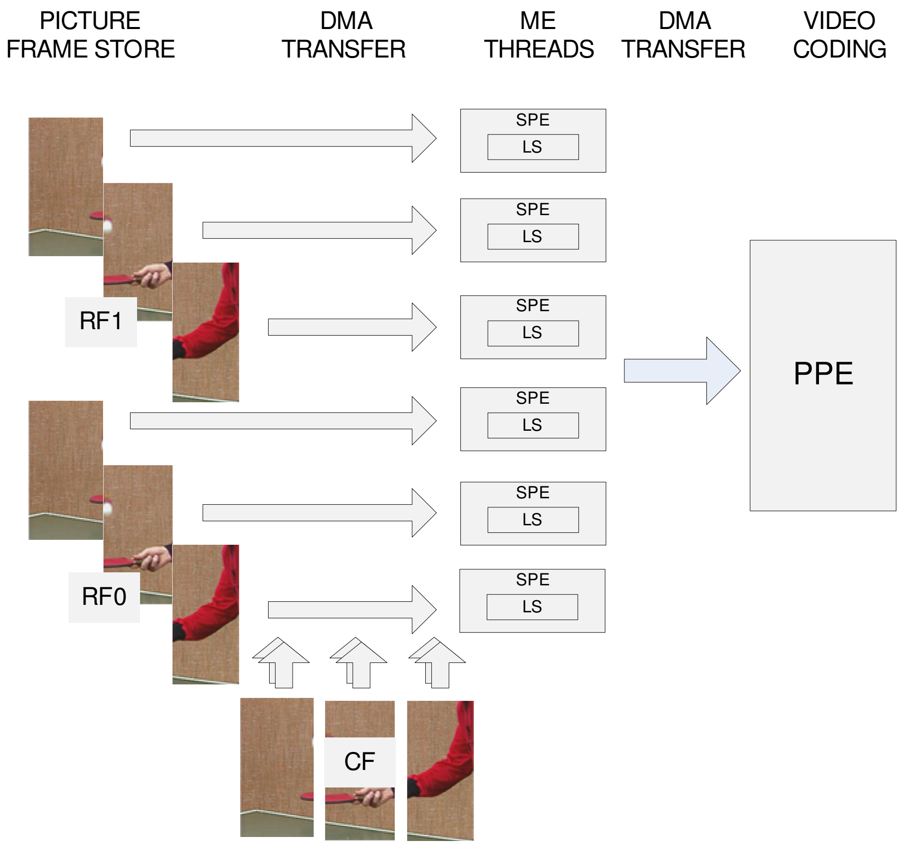


FIG. 2.6. *Scalable parallelization approach.*

achieved by using mailboxes. By a matter of organization, the PPE program is presented first, followed by the SPE program, and at the end, the way of using the SIMD instruction extensions to exploit further parallelism and additionally speed up is given.

3.2. PPE. The program in the PPE executes Algorithm 1. Initialization of the program includes: definition and calculation of the required parameters, allocation of memory for the buffers, and creation of the threads. General algorithm parameters are defined, such as the number of threads, the number of vertical slices, width in MBs of each vertical slice, search area range, search algorithm that will be performed etc. After

Algorithm 1 PPE side of the algorithm

```
1:  allocate buffers, create and initialize threads
2:  calculate the required number of vertical slices
3:  for each P frame and row do
4:    if row is first and frame is first then
5:      ppu_msg = $NEW\_FRAME$
6:    else
7:      ppu_msg = $NEW\_ROW$
8:    end if
9:    if row is not last then
10:     for each vertical slice and RF do
11:       fill the buffer of the related thread
12:     end for
13:     send ppu_msg to threads
14:   end if
Ensure:  confirmation is received
15: end for
16: send KILL message to threads
```

that, *pels*0, *pels*1 and *results* buffers are allocated for each thread, the threads are created, and, finally, initial parameters are passed to the SPEs. The addresses of the related buffers are also sent to the SPEs.

For each P frame the following actions are performed: *i)* A row of MBs/SAs is loaded into the buffers corresponding to the different vertical slices and RFs; *ii)* when the buffers are filled a message is passed to each thread in order to start the DMA transfer for the next row or the next frame if the last row of the previous frame was already transferred. The reason for distinction between NEW_ROW and NEW_FRAME messages is that the SA chunk for the initial row in a frame has a different size, as it was shown in Figure 2.3; and *iii)* the PPE waits for the ME results and for the confirmation message from each thread that a row is finished. The time passed since entering in the loop till the confirmation message is received corresponds to the time that VC has to wait for ME to be finished, and it is called ME row delay. By summing all the ME row delays in a frame we get the ME time for a single frame. When all P frames are examined, a KILL message is sent to SPE threads.

3.3. SPE. SPE threads implement the search algorithm according to the steps represented in Algorithm 2. The algorithm is divided in three main parts: *i)* the first one performs direct communication with the PPE, *ii)* the second part is responsible for the buffers' management, and *iii)* the final part implements the search algorithm.

The first part initializes the process and starts the infinite loop, waiting for the PPE commands. The initialization part includes allocation of the buffers, namely *pels*0, *pels*1 and *results*, and the registration of the DMA addresses, namely *pels*0*_addr*, *pels*1*_addr* and *results_addr*. These addresses are set up with the corresponding *pels*0, *pels*1 and *results* buffer lines, on the PPE side. The NEW_FRAME, NEW_ROW or KILL messages are expected in mailboxes during the execution of the infinite loop. If NEW_FRAME or NEW_ROW messages are received, the algorithm reaches the first synchronization point, which means that the pixels of the current row are loaded. Then the SPE starts the loading of the next row pixels in parallel with the processing of the current row. If it is the initial row, an initial delay has to be introduced, because no data exists to be processed in parallel. In the case of NEW_FRAME message all SA pixels in a row have to be loaded, while in the case of the NEW_ROW message, just the new pixels are loaded while the ones that belong to the previous SA row are reused, (see Figure 2.3). When the ME of current row is finished, the results are sent back to the PPE.

The third part (line 16) corresponds to the implementation of the search procedure. The search procedure is not the aim of this work, so the proposed parallelization approach considers the implementation of both computationally intensive and robust search algorithms (requiring less computation but more memory). After processing a whole MB row, the content of the *result* buffer is passed to the main memory (PPE) by a DMA transfer. When the transfer is finished, a confirmation message is sent to the PPE notifying that the results are available and can be used by VC. The algorithm is finished when a KILL mail is received.

3.4. Using specialized SIMD instructions. The CELL architecture offers a set of vector instructions in both PPE and SPE. In the proposed encoder, these available instructions are used to compute in parallel the SAD defined in eq. 3.1, for different pixels:

$$SAD(v_x, v_y) = \sum_{m,n=0}^{N-1} |CF(x+m, y+n) - RF(x+v_x+m, y+v_y+n)|, \tag{3.1}$$

where (v_x,v_y) are coordinates of the MV referring to the SA and (x,y) are the coordinates of the left-up angle of the considered MB referring to the frame.

As it is well known, data misalignment is one of the main constraints when using vector instructions. The SPE loads and stores only support quadword (16-byte) aligned data, masking the 4 least significant bits of the address. However, search algorithms for ME run in a pixel-by-pixel basis, therefore requiring pixel based (misaligned) data access and calculation. In the proposed data alignment scheme presented in Figure 3.1 data is packed in aligned byte element vectors using the instructions from the SPE SIMD intrinsics library. Specialized $spu\_shuffle$ intrinsic combines the bytes of two vectors, according to the organization defined in a third vector. The required alignment patterns are stored in a pre-calculated look-up table in SPE LS. Two typical aligning situations of both misaligned MB and SA are presented in Figure 3.1 *a)* and *b)*, respectively. Each MB line is placed in a single 16Bytes area, because the first MB starts from an aligned position, and the MB width of 16 pixels is used. Therefore, the sub-block lines can only be "shifted" inside of a single 16Bytes aligned area. However, SAs' vectors can be divided between two successive 16Bytes areas, and in this case one additional $spu\_shuffle$ instruction has to be used.

Figure 3.1 *c)* depicts the SAD value calculation. The scheme is similar to the one proposed in [9]. First the absolute difference (*spu_ad* intrinsic) vectors are calculated for the SA and MB values packed in byte vectors. Then the *spu_sumb* intrinsic instruction is applied in order to sum each 4 elements of the vectors in 16bit result values. Finally, the obtained values are accumulated in the SAD vector.

The same SIMD processing scheme is adopted in the PPE, with the exception that the luminance of the pixels in the JM 14.0 encoder is stored as *unsigned short* values, which has a negative impact in the obtained speedup.

Algorithm 2 SPE side of the algorithm

```
 1: allocate buffers
 2: read buffer addresses
 3: repeat
 4:   read in mailbox
 5:   if $msg = KILL$ then
 6:     end algorithm
 7:   end if
 8:   if $msg = NEW\_FRAME$ or $msg = NEW\_ROW$ then
 9:     if $msg = NEW\_FRAME$ then
10:       increase DMA transfer size
11:     end if
12:     if row is initial then
13:       load first row pixels
14:     end if
Ensure:     DMA transfer of current row pixels is finished
15:     start loading of next row
16:     process current row
17:     send results and confirmation
Ensure:     DMA transfer of results is finished
18:   end if
19: until true
```

4. Experimental Results. The experimental results are obtained by using the JM 14.0 software implementation of H.264/MPEG-4 AVC. Seven different sub-block types and 5 RFs are considered. The optimal FSBM and adaptive Unsymmetrical-cross Multihexagon-grid Search (UMHS) search algorithms are both programmed. All frames except the first one are coded as P-frames, with the ME search range of 16 pixels, and a quantization step QP=28. Both Common Intermediate Format (CIF), 352×288 pixels, and HD, 720×576 pixels, test video sequences are used. Sequences *akiyo*, *bus* and $foreman$ for CIF and $blue\_sky$, $riverbed$ and $rush\_hour$ for HD are chosen because of their different characteristics regarding motion and spatial details. The presented

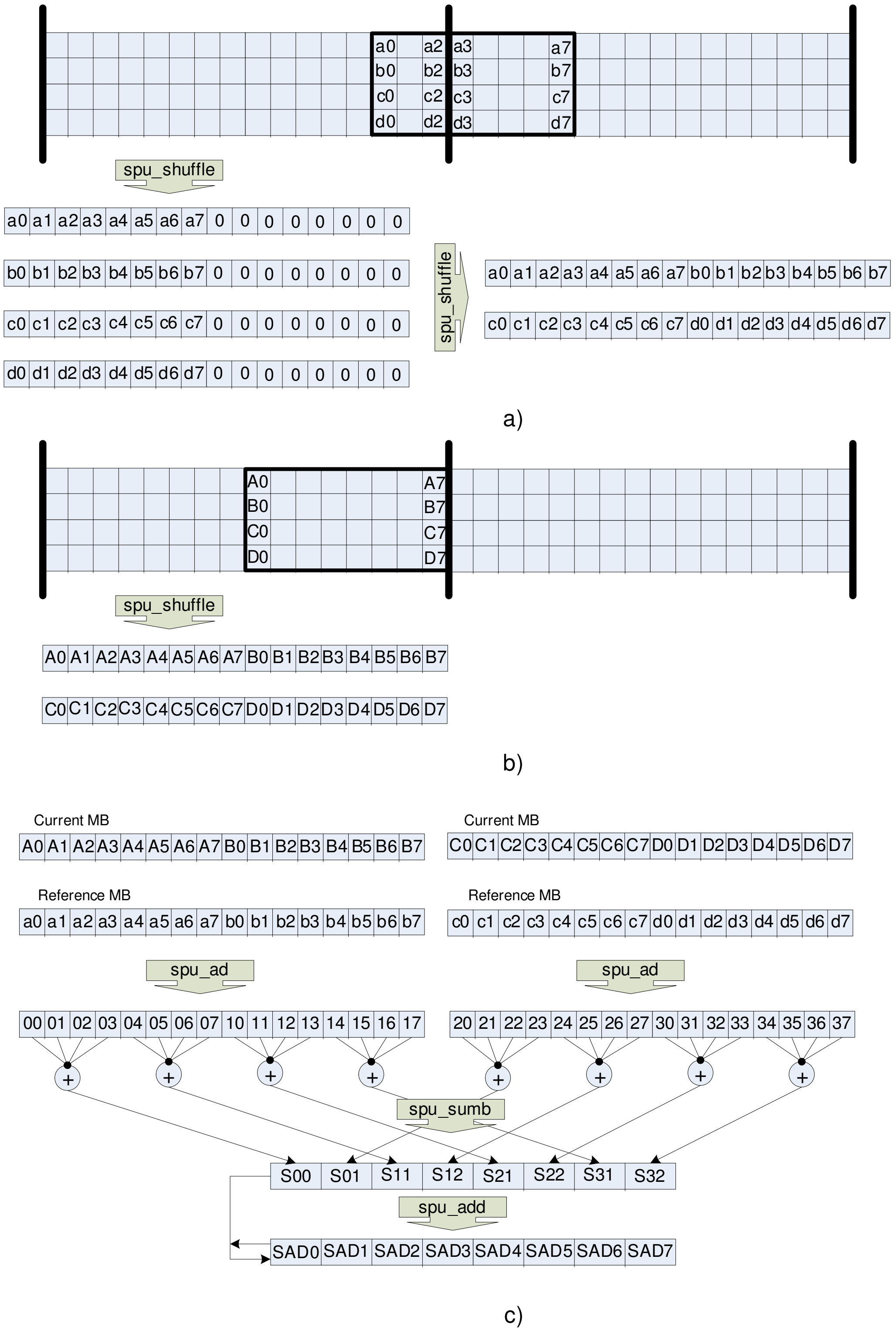


FIG. 3.1. *Data alignment and SAD calculation process on the SPEs: a) MB alignment b) SA alignment c) SAD calculation*

results correspond to average values for 50 frames. In addition to the described implementation, experimental results are obtained for ME implemented in the PPU, also using the AltiVec SIMD extensions and a Dual core AMD Opteron 170 cental processor unit (CPU), with 2GBytes of main memory and SuSe 10.3 operating system.

Table 4.1
ME time (ms) per frame for the different systems and search algorithms

	CELL		CPU		PPU	
	UMHS	FSBM	UMHS	FSBM	UMHS	FSBM
akiyo CIF	18	349	352	3995	532	2382
bus CIF	79	348	695	4001	893	2356
foreman CIF	48	348	502	3998	703	2372
blue_sky 720×576	205	1530	2105	16859	3357	10151
riverbed 720×576	286	1530	3117	16842	4238	10172
rush_hour 720×576	188	1530	1870	16863	2179	10191

Table 4.2
Required SPU memory in kBytes

	Program	Data	Total
without SIMD instr.	14.1	4	18.5
with SIMD instr.	15.5	5.5	21
UMHexagonS+SIMD	16.5	6.0	22.5

Table 4.1 presents the time required for ME in the Cell processor (proposed one), as well as in the PPU and the CPU; for both UMHS and FSBM algorithms. Results prove the efficiency of the multi-core solution compared with single core solutions on CPU and PPU. Even without partially hiding the ME behind the VC, the real time ME for the CIF format is nearly achieved, when UMHS algorithm is used.

In figure 4.1 the four different charts show the scalability of the proposed algorithm regarding the independence between the number of threads and the number of RFs used, as well as video sequence resolution, when 6 SPUs are used. In the cases when less than 6 RFs are used, all the CF and the RFs are divided in vertical slices, namely 2, 3 and 6 slices, for 3, 2 and 1 RFs, respectively.

Figures 4.1 a) and c) show results for the optimal FSBM algorithm but with different resolutions. As it is shown in the charts for both resolutions a regular ME time decrease is obtained using the parallel algorithm with the reduction of RFs, or, what is equivalent, with the increase in the number of vertical slices considered for each frame.

Figures 4.1 b) and d) show results for the UMHS search algorithm. In both charts it is visible a trend of ME time decrease, with increase in number of vertical slices that each frame is divided in (inversely proportional to the number of RFs). A faster ME time decrease with the increase in the number of vertical slices when the motion is spatially more balanced, as in the *bus* and *riverbed* sequences, is also visible. In the case of *riverbed* sequence the curve evolution is very similar to the one for the FSBM algorithm, because motion is quite well spatially balanced.

Figure 4.2 presents the time that the remaining part of VC has to wait for the ME to be finished, which includes preparation of the buffers and communication between the PPU and SPUs. The ME for the next MB row is running in parallel with the VC of the current MB row, and, therefore, ME is almost hidden behind the VC time.

Figure 4.3 shows that ME is no longer the bottleneck of video coding if the proposed encoder is used. In the case when either PPU or CPU is used, ME takes up to 95% of encoding time, as it was already reported, but in the presented approach it takes less than 15%. Results are even better if HD video sequences are used, because VC time increases more than the ME delay.

Table 4.2 presents the SPU memory required by the proposed motion estimator. The values show that a very small part of the 256kByte total LS is used (less than 15%), and therefore there is plenty of space for fast algorithms development, which can require a lot of memory. The presented result for UMHS includes only spatial prediction. If a full version with the temporal prediction is used the MVs and distortion measures for multiple frames need to be stored, and up to 30kBytes per prediction frame is required for temporal prediction [3], if CIF format is considered.

5. Conclusions. Starting from a parallel algorithm proposed by the author of this paper, a scalable ME parallel algorithm was proposed. This algorithm supports a variable number of cores and RFs achieved by

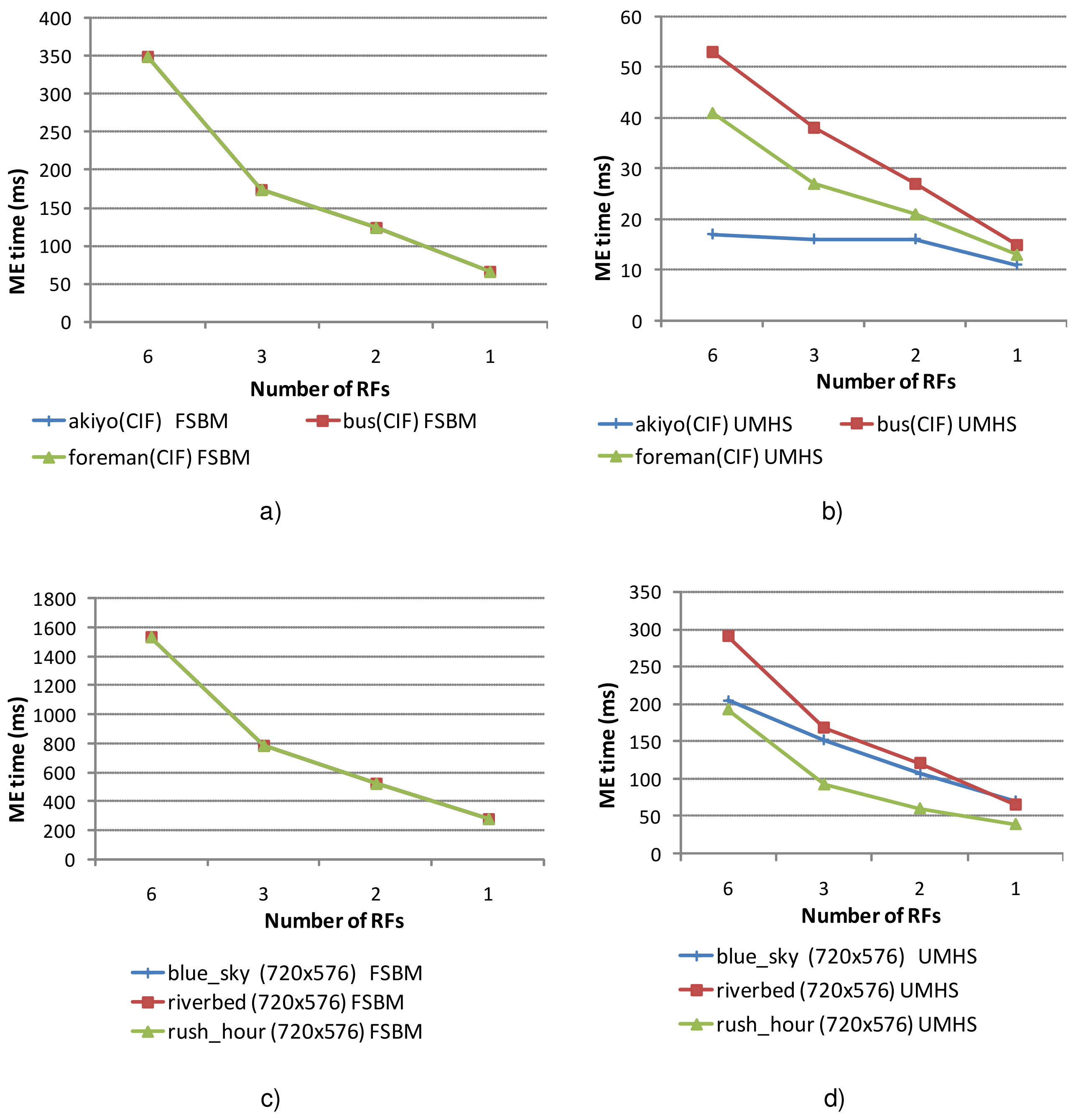


Fig. 4.1. *Time(ms) per frame for 6 SPUs, considering different number of RFs, video formats and search algorithms: a) CIF video format and FSBM b) CIF video format and UMHS c) 720×526 resolution and FSBM b) 720×526 resolution and UMHS*

subdividing CF and RFs into vertical slices, according to the number of cores available and the number of RFs required. The parallel algorithm was implemented in a STI CELL Broadband Engine Processor, using SIMD extensions to compute multiple SADs in parallel. To evaluate the efficiency of the proposed model, H.264/MPEG-4 AVC motion estimation, with 5 RFs and 7 different MB shapes was considered. Experimental results are presented for both CIF and HD video formats. For comparison purposes, both results for ME on PowerPC(labeled as a PPU) and Dual Core PC are also presented. Results show that the proposed algorithm is scalable, regarding the video resolution, the number of cores and the number of used RFs. In this work only P

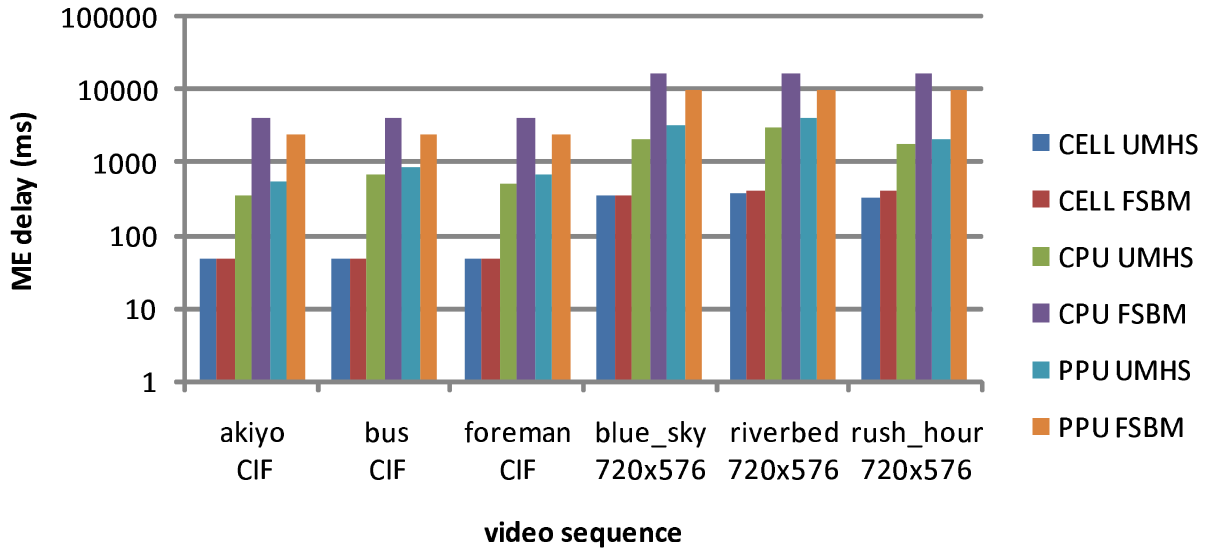


Fig. 4.2. *ME delay (ms) per frame for the different implementations of ME.*

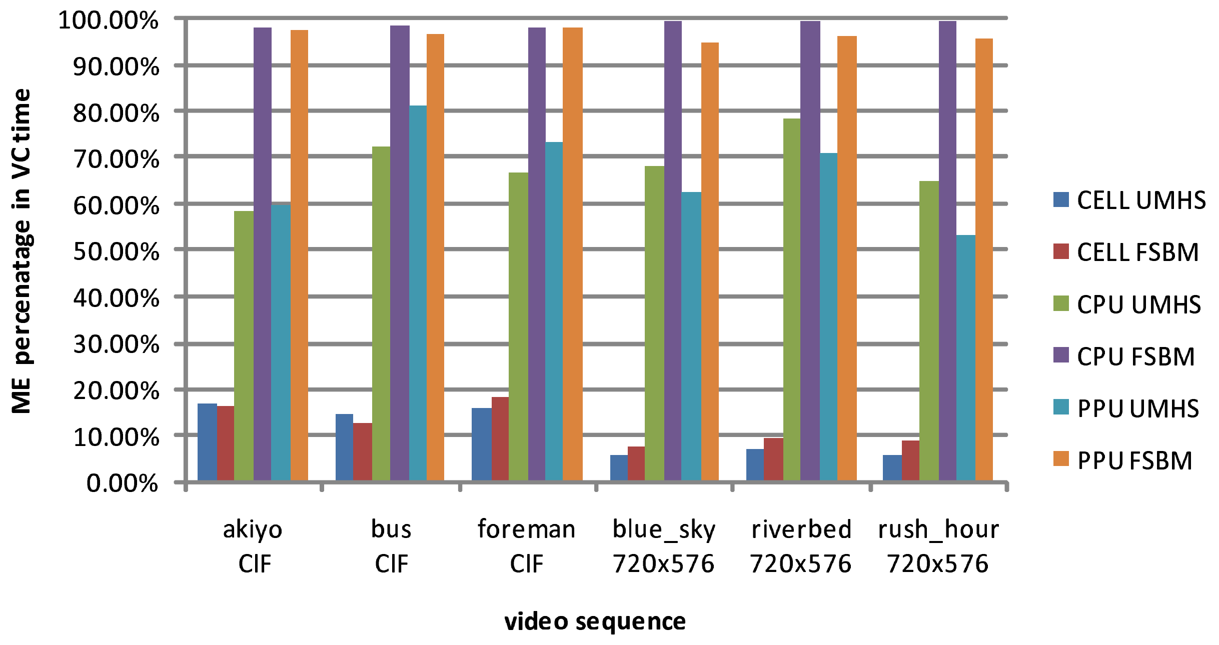


Fig. 4.3. *Percentage in total encoding time for the different motion estimators.*

frames and full pixel accuracy are considered. Moreover, it is shown that the CELL processor is able to perform ME in real time for CIF images, if the optimal full search approach is adopted.

Acknowledgments. This work has been supported by the POSI program and the Portuguese Foundation for Science and for Technology (FCT) under the research project Adaptive H.264/AVC Motion Estimation Processor for Mobile and Battery Supplied Devices (AMEP) POSI/EEA-CPS/60765/2004.

REFERENCES

[1] J. Ostermann et al, *Video coding with H.264/AVC: tools, performance, and complexity*, IEEE Circuits and Systems Magazine, vol. 4, pp. 7–28.

[2] S.-Y. Huanga and W.-C. Tsai, *A simple and efficient block motion estimation algorithm based on full-search array architecture*, Signal Processing: Image Communication, vol. 19, pp. 975–992, 2004.

[3] Z. Chen et al, *Fast integer-pel and fractional-pel motion estimation for* H.264/AVC, Journal of Visual Communication and Image Representation, October 2005, pp. 264–290.

[4] S. Momcilovic, N. Roma, and L. Sousa, *An* ASIP *approach for adaptive motion estimation on* AVC, Proceedings of the IEEE 3rd Conf. on Ph.D. Research in Microelectronics and Electronics (PRIME 2007), Bordeux, France, July 2007, pp. 165–168.

[5] J.-C. Chu et al, *An embedded conherent-multithreading multimedia preocessor and its programming model*, Proceedings of the 44th Design Automation Conference (DAC 2007), San Diego, California, USA, June 2007, pp. 652–657.

[6] C. R. Johns and D. A. Brokenshire, *Introduction to the cell broadband engine architecture*, IBM Journal of Research and Development, vol. 51, no. 5, pp. 503–519, September 2007.

[7] Sony Computer Entertainment Incorporated, SPU C/C++ *Language Extensions*, 2005.

[8] L.-K. Lui et al, *Video alalysis and compression on the* STI CELL *broadband engine*, Tech. Rep.

[9] L. B. Shyang, *A Simplified High Definition Video Encoder Based on The* STI CELL *Multiprocessor*, Student thesis, Institutionen för systemteknik, Linköping, Sweden, January 2007.

[10] S. Momcilovic and L. Sousa, *An parallel algorithm for advanced video motion estimation on multicore architectures*, Proceedings of the 2nd International Conf. on Complex, Intelligent and Softare Intensive Systems (CISIS 2007), Barcelona, Spain, March 2008, pp. 831–836.

[11] International Business Machines (IBM) Corporation, CELL *Broadband Engine Programming Tutorial*, 2006.

[12] M. Gschwind, *The* Cell Broadband Engine*: Exploiting multiple levels of parallelism in a chip multiprocessor*, International Journal of Parallel Programming, vol. 35, pp. 233–262, 2007.

[13] JVT *Reference Software unofficial version 12.0*, `http://iphome.hhi.de/suehring/tml/download`

Edited by: Sabri Pllana, Siegfried Benkner
Received: June 16, 2008
Accepted: July 28, 2008

Scalable Computing: Practice and Experience
Volume 9, Number 3, pp. 219–220. http://www.scpe.org

ISSN 1895-1767
© 2008 SCPE

BOOK REWIES

EDITED BY SHAHRAM RAHIMI

Artificial Intelligence: Structures and Strategies for Complex Problem Solving
by George F. Luger
6th edition, Addison Wesley, 2008

The book serves as a good introductory textbook for artificial intelligence, particularly for undergraduate level. It covers major AI topics and makes good connection between different areas of artificial intelligence. Along with each technique and algorithm introduced in the book, is a discussion of its complexity and application domain. There is an attached website to the book that provides auxiliary materials for some chapters, sample problems with solutions, and ideas for student projects. Besides Prolog and Lisp, java and C++ are also used to implement many of the algorithms in the book.

The book is organized in five parts. The first part (chapter 1) gives an overview of AI, its history and its various application areas. The second part (chapters 2–6) concerns with knowledge representation and search algorithms. Chapter 2 introduces predicate calculus as a mathematical tool for representing AI problems. The state space search as well as un-informed and heuristic search methods is introduced in chapters 3 and 4. Chapter 5 discusses the issue of uncertainty in problem solving and covers the foundation of stochastic methodology and its application. In chapter 6 the implementation of search algorithms is shown in production system and blackboard architectures.

Part 3 (chapters 7–9) discusses knowledge representation and different methods of problem solving, including strong, weak and distributed problem solving. Chapter 7 begins with reviewing the history of evolution of AI representation schemes, including semantic networks, frames, scripts and conceptual graphs. This chapter ends with a brief introduction of Agent problem solving. Chapter 8 presents the production model and rule-based expert systems as well as case-based and model-based reasoning. The methods of dealing with various aspects of uncertainty are discussed in chapter 9. These methods include Dempster-Shafer theory of evidence, Bayesian and Belief networks, fuzzy logics and Markov models.

Part 4 is devoted to machine learning. Chapter 10 describes algorithms for symbol-based learning, including induction, concept learning, vision-space search and ID3. The neural network methods for learning, such as back propagation, competitive, Associative memories and Hebbian Coincidence learning were presented in chapter 11. Genetic algorithms and evolutionary learning approaches are introduced in chapter 12. Chapter 13 introduces stochastic and dynamic models of learning along with Hidden Markov Models, Dynamic Baysian networks and Markov Decision Processes.

Part 5 (chapters 14 and 15) examines two main application of AI: automated reasoning and natural language understanding. Chapter 14 begins with an introduction to weak methods in problem solving and continues with presenting resolution theorem proving. Chapter 15 deals with the complex issue of natural language understanding by discussing main methods of syntax and semantic analysis of natural language corpus. The chapter ends with examples of natural language application in Database query generation, text summarization and question answering systems. Finally, chapter 16 is a summary of the materials covered in the book as well current AI's limitations and future directions.

One criticism about the book would be that the materials are not covered in enough depth. Because of the space limitation, many important AI algorithms and techniques are discussed briefly without providing enough details. As a result, some chapters (e.g., 8, 9, 11, and 13) of the book should be supported by complementary materials to make it understandable for undergraduate students and motivating for graduate students. Another issue is with the structure of the book. The order of presenting chapters introduces sequentially different challenges and techniques in problem solving. Consequently, some topics such as uncertainty and logic are not introduced separately and are distributed in different chapters of the book related to different parts. Although interesting, this makes the book hard to follow.

In summary, the book gives a great insight to the readers that want to familiar themselves with artificial intelligence. It covers a broad range of topics in AI problem solving and its practical application and is a good reference for an undergraduate level introductory AI class.

Elham S. Khorasani,
Department of Computer Science
Southern Illinois University
Carbondale, IL 62901, USA

AIMS AND SCOPE

The area of scalable computing has matured and reached a point where new issues and trends require a professional forum. SCPE will provide this avenue by publishing original refereed papers that address the present as well as the future of parallel and distributed computing. The journal will focus on algorithm development, implementation and execution on real-world parallel architectures, and application of parallel and distributed computing to the solution of real-life problems. Of particular interest are:

Expressiveness:
- high level languages,
- object oriented techniques,
- compiler technology for parallel computing,
- implementation techniques and their efficiency.

System engineering:
- programming environments,
- debugging tools,
- software libraries.

Performance:
- performance measurement: metrics, evaluation, visualization,
- performance improvement: resource allocation and scheduling, I/O, network throughput.

Applications:
- database,
- control systems,
- embedded systems,
- fault tolerance,
- industrial and business,
- real-time,
- scientific computing,
- visualization.

Future:
- limitations of current approaches,
- engineering trends and their consequences,
- novel parallel architectures.

Taking into account the extremely rapid pace of changes in the field SCPE is committed to fast turnaround of papers and a short publication time of accepted papers.

INSTRUCTIONS FOR CONTRIBUTORS

Proposals of Special Issues should be submitted to the editor-in-chief.

The language of the journal is English. SCPE publishes three categories of papers: overview papers, research papers and short communications. Electronic submissions are preferred. Overview papers and short communications should be submitted to the editor-in-chief. Research papers should be submitted to the editor whose research interests match the subject of the paper most closely. The list of editors' research interests can be found at the journal WWW site (`http://www.scpe.org`). Each paper appropriate to the journal will be refereed by a minimum of two referees.

There is no a priori limit on the length of overview papers. Research papers should be limited to approximately 20 pages, while short communications should not exceed 5 pages. A 50–100 word abstract should be included.

Upon acceptance the authors will be asked to transfer copyright of the article to the publisher. The authors will be required to prepare the text in $\LaTeX 2_\varepsilon$ using the journal document class file (based on the SIAM's `siamltex.clo` document class, available at the journal WWW site). Figures must be prepared in encapsulated PostScript and appropriately incorporated into the text. The bibliography should be formatted using the SIAM convention. Detailed instructions for the Authors are available on the SCPE WWW site at `http://www.scpe.org`.

Contributions are accepted for review on the understanding that the same work has not been published and that it is not being considered for publication elsewhere. Technical reports can be submitted. Substantially revised versions of papers published in not easily accessible conference proceedings can also be submitted. The editor-in-chief should be notified at the time of submission and the author is responsible for obtaining the necessary copyright releases for all copyrighted material.