University of Delaware
Department of Electrical and Computer Engineering
Computer Architecture and Parallel Systems Laboratory

Toward Efficient Fine-grained Dynamic Scheduling on Many-Core Architectures

Elkin Garcia
Daniel Orozco
Robert Pavel
Guang R. Gao

CAPSL Technical Memo 111
February, 2012
Revised, March 2012

Copyright © 2012 CAPSL at the University of Delaware

{egarcia@, orozco@, rspavel@, ggao@capsl.}udel.edu

Contents

List of Figures

List of Tables

Abstract

The recent evolution of many-core architectures has resulted in chips where the number of processor elements (PEs) are in the hundreds and continue to increase every day. In addition, many-core processors are more and more frequently characterized by the diversity of their resources and the way the sharing of those resources is arbitrated.

On such a machine, task scheduling is of paramount importance to orchestrate a satisfactory distribution of tasks with an efficient utilization of resources, especially when fine-grain parallelism is desired or required. In the past, the primary focus of scheduling techniques has been on achieving load balancing and reducing overhead with the aim to increase total performance. This focus has resulted in a scheduling paradigm where Static Scheduling (SS) is preferred over Dynamic Scheduling (DS) for highly regular and embarrassingly parallel applications running on homogeneous architectures.

We have revisited the task scheduling problem for these types of applications under the scenario imposed by many-core architectures to investigate whether or not there exists scenarios where DS is better than SS.

Our main contribution is the idea that, for highly regular and embarrassingly parallel applications, DS is preferable over SS in some situations commonly found in many-core architectures. We present experimental evidence that shows how the performance of SS is degraded by the new environment on many-core chips.

We analyze three reasons that contribute to the superiority of DS over SS on many-core architectures under the situations described:

1. A uniform mapping of work to processors without considering the granularity of tasks is not necessarily scalable under limited amounts of work.
2. The presence of shared resources (i.e. the crossbar switch) produces unexpected and stochastic variations on the duration of tasks that SS is unable to manage properly.
3. Hardware features, such as in-memory atomic operations, greatly contribute to lower the overhead of DS.

1 Introduction

Recent trends in computer architecture, where manycore processors are routinely composed of hundreds of processing cores, has unleashed challenges in many aspects of computing technology. Task scheduling, in particular, is difficult in manycore architectures due to the quantity and diversity of resources available: Static Scheduling (SS) was usually preferred over Dynamic Scheduling (DS) for regular, embarrassingly parallel applications on general purpose architectures. However, SS is not necessarily the best choice for manycore architectures, even for regular, embarrassingly parallel applications.

The two main factors that usually hurt the expected advantages of SS over DS in many-cores are: 1) The large number of processing elements in a many-core chip results in fewer tasks assigned to each processing element, and 2) the behavior and interaction of shared resources is not necessarily uniform during execution. These two new factors blunt the effectiveness of SS while greatly favoring DS, even under scenarios where SS has traditionally been the logical solution.

Shared resources are also an important source of task imbalance because the arbitration of shared resources may produce unexpected stalls that could change the completion time of similar tasks. The most common shared resources on many-cores are the memory, the communication infrastructure (e.g. crossbar, access ports), and the functional units (e.g. Floating Point units and other special purpose units).

All of these sources of imbalance make it difficult for SS to provide a strategy that fully utilizes the hardware, producing results below those expected, even for regular, classical applications like Matrix Multiply [1].

The nature of DS can manage and compensate for the unpredictable effects of resource sharing and imbalance introduced by the granularity of tasks. However, it remains a challenge to execute a low-overhead implementation of DS in architectures without adequate hardware support. In contrast, when hardware support is available, it is possible to deliver high throughput and low latency using a DS implementation with overall results superior to those of an SS approach. This superiority can be observed in situations that were traditionally favorable to SS (regular applications in homogeneous architectures) and in situations with fine-grained tasks with some degree of heterogeneity.

The paper is organized as follows: Section 2 presents a motivating example using Matrix Multiplication as well as providing some required background. Section 3 defines the problem addressed in this paper. Section 4 shows the impact of work partitioning and shared resources in SS and DS. This section also analyzes the overhead of the light DS implementation proposed. Section 5 presents our results on micro-benchmarks and real applications, talking about the effectiveness of our approach. Section 6 presents other related work in the field. Finally, Section 7 presents our conclusions and possible directions for our future work.

Operand Placement	Optimization	Performance (GFLOPS)
SRAM	Static Partition	3.16
SRAM	+Register Tiling	30.42
SRAM	+Instruction Scheduling	44.20
DRAM	Percolation +Synch. Optimization +Opt. Data Movement	13.90

Table 1: Summary of Previous Results of MM on C64

2 Background and Motivation

Several efforts have been made to study the optimization of applications[1, 2, 3, 4] for the IBM Cyclops-64 (C64)[5]. Those studies, however, show results that are still far from the maximum theoretical performance. To understand the problem, we have analyzed one of the simplest examples of Static Scheduling to find the issues that have prevented better results.

Cyclops-64 (C64) is a manycore architecture designed for High Performance Computing (HPC). A C64 chip contains 160 independent single-issue thread units (TUs), up to 4.8MB of shared on-chip memory (SRAM) and 1GB of external memory (DRAM). Each pair of TUs shares one 64-bit floating point unit (FPU), one memory bank and a memory controller. The FPUs can fire 1 double precision floating point *Multiply and Add* instruction per cycle for a total performance of 80 GFLOPS per chip when running at 500MHz. A 96-port crossbar network with a bandwidth of 4GB/s per port connects all TUs and SRAM banks. Execution on a C64 chip is non-preemptive and there is no hardware virtual memory manager.

Two recent studies in the implementation of Matrix Multiplication on C64 [6, 1] have shown the effectiveness of several optimization techniques (Table 1). The initial implementations studied targeted SRAM and DRAM and it achieved a performance of 13.9 GFLOPS [6]. Further optimization of on-chip SRAM memory usage resulted in a performance of 44.12 GFLOPS [1].

As can be observed from Table 1, the maximum performance reported after several static optimizations barely surpassed one half of the peak performance when all the operands are in SRAM.

The implementation in our study (Matrix Multiply) was improved to 58.95 GFLOPS when Optimum Register Tiling [1] with Data-Prefetching was used. Surprisingly, this implementation is still far from the expected peak performance, even after months of optimizing carefully-written assembly code, and after significant theoretical and experimental effort to find an optimal register-tiling strategy [1].

The comparatively low performance achieved – even after the carefully optimized assembly code implementation– prompted an investigation into the factors that prevented us from reaching a higher performance. To do so, we conducted an extensive and careful profiling of

the application.

Two cases, both using SS, were studied in particular: A multiplication of the largest matrices that can fit in SRAM (Figure 1(a)) and a multiplication of smaller matrices also in SRAM (Figure 1(b)). Smaller matrices are required for implementation of matrix multiplication in DRAM doing overlapping of computation and data movement with SRAM.

The following observations can be made:

- The amount of time computing tiles whose size was optimized for maximum performance does not surpass 70% of the execution time in any thread.
- The problem size is not always a multiple of the optimum tile size, so smaller tiles have to be included in the computation, adding to the imbalance of the system. This causes problems because either (1) the computation is partitioned into tiles, which may result in different number of tiles per thread or (2) the computation is partitioned evenly among threads, which may not be as efficient as partitioning the computation in carefully chosen tiles.
- In general, when the size of a problem decreases, the fraction of tiles that are not of the best size increases. This hurts the performance because smaller tasks may not fully take advantage of the available resources, even if they are optimized.
- The idle (wasted) time includes the time spent in synchronization, flow control, and scheduling. It is more than 10% in the best case and increases dramatically when the matrix size decreases.

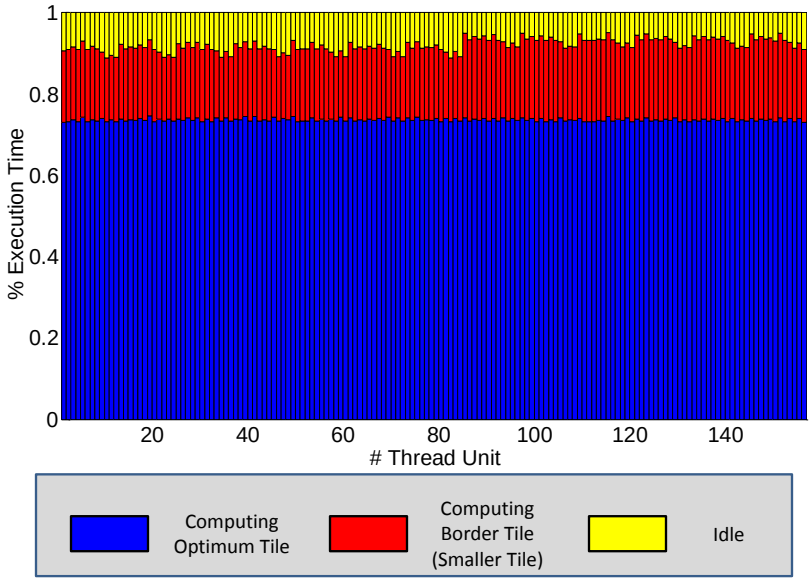
We see that SS is not necessarily a good strategy in manycore processors, even for the case of highly regular parallel benchmarks.

3 Problem Statement

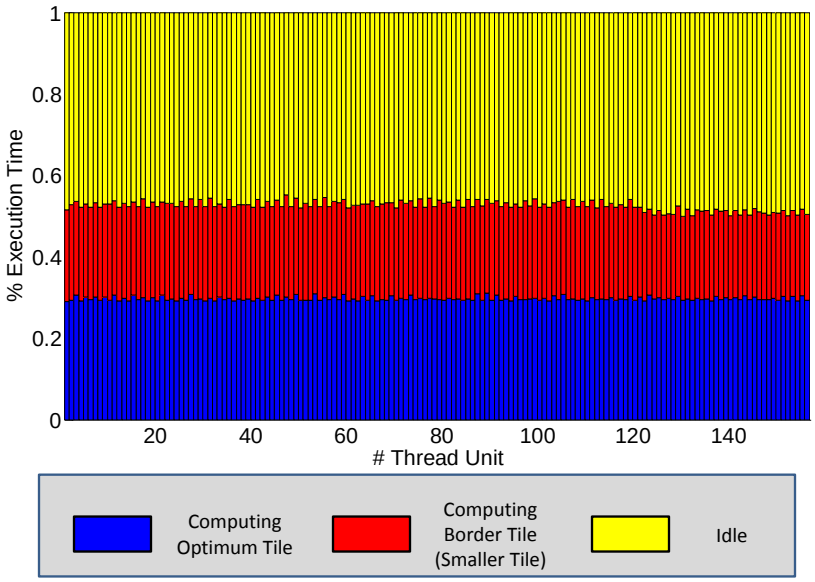
The following question summarizes our research goals:

How is it possible to efficiently schedule fine grain tasks in the case of scenarios with hundreds of threads sharing multiple resources under different conditions, such as varying degrees of utilization of available resources? In particular,

- Is it possible to reach near peak performance on many-cores in situations where there is a high demand for available resources?
- Which elements are necessary to implement for an efficient fine-grain scheduler?
- Is it possible to implement an efficient Dynamic Scheduler that surpasses the results of a Static Scheduler, even under scenarios traditionally favorable to SS?
- How can the overhead of DS be overcome during the scheduling of fine-grained tasks?



(a) Workload Distribution for a MM of size 488×488

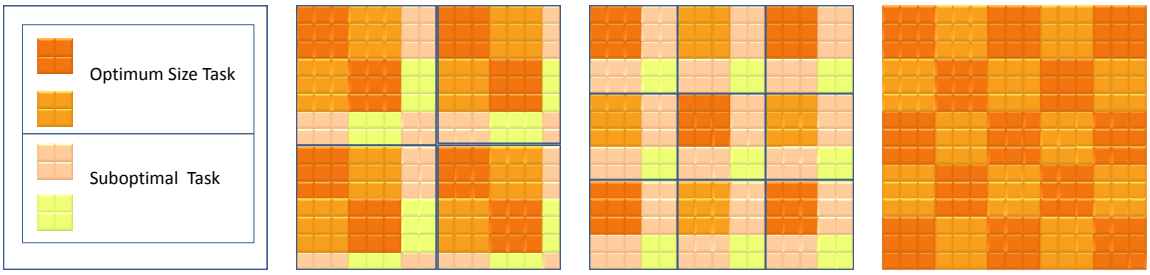


(b) Workload Distribution for a MM of size 192×192

Figure 1: Workload Distributions for MMs of varying sizes

4 Dynamic Scheduling for fine grained parallelism

Finer task granularity is one of the ways in which enough parallelism is provided for manycore processors. We strive to address these challenges by making observations that motivate a



(a) Legend (b) SS for $PE = 4$ (c) SS for $PE = 9$ (d) Without blocks

Figure 2: Partition Schemes for a matrix C of 15×15 with tiles of 3×3

deeper understanding of the trade-offs that are normally left unconsidered by SS, even in the face of new many-core architectures. We will show how DS is a better alternative due to the disadvantages of SS, even in the presence of fine-grained tasks. First, we will discuss the impact of a fine-grained task partition on overall performance. Second, we will show the implications on load balancing and performance in a scenario where resources are shared between PEs. Third, we will study the required characteristics for a low overhead DS and how to implement it efficiently for a set of similar tasks.

4.1 Fine-grained task partitioning

In this section, we will explain that, under certain conditions, DS can result in faster execution of programs because it can partition the work into better tasks. At this point we consider ideal conditions of no scheduling overhead, no shared resources, and tasks with very similar execution times. On the other hand, SS results in a work partitioning that may or may not be the better choice for the architecture used. We will expand this reasoning to include more realistic scenarios in the following sections.

The problem faced by SS is the trade-off between load balance and the efficient processing of tasks given by the partition of data into blocks for threads and the further partitioning of blocks into tasks for optimal execution. On one hand, an SS that maximizes load balancing will distribute equally sized blocks among all processors. Unfortunately, the partitioning of blocks may result in non-optimum tiles at the boundaries of blocks, decreasing performance. This is even worse for situations where the ratio between the block size and task size decreases due to a limited amount of on-chip memory or an increasing number of processing units results in smaller tasks. These two cases are particularly evident on modern many-core architectures. Figure 2 illustrates different scenarios for the amount of data in border tiles (highlighted). The worst scenario is when number of processing elements (PEs) is increased and the best one is when blocks are not used. On the other hand, an SS that uses tasks will decrease the penalty due to border tiles but will decrease load balancing in cases where the number of tasks is not a multiple of the number of processing elements.

In contrast, DS has the ability to deal with these unbalanced scenarios that decrease the size

of border tiles. The partition of data into blocks is not required and each processing element request a new task as soon as it finishes the previous one. Assuming a similar overhead of SS and DS, DS will be able to produce equal or better scheduling than SS given the fact the assignation of task is given at run time. Further analysis of how to deal with shared resources and how to reach a competitive overhead on DS will be discuss in the following sections.

4.2 Load Balancing in Shared Resources Scenarios

We have analyzed an ideal scenario where tasks of the same size would take the same time to complete. Unfortunately, this does not take into account the indirect interaction between tasks given by shared resources involving arbitration of third parties and starvation.

Some functional units are shared in order to diminish the required chip area and power consumption, saving room so as to include more PEs that can increase the overall performance. Examples are Floating Point Units or special purpose units, such as specialized DSP blocks. Sharing these resources impose limitations, especially in SIMD programs. While one PE is using the shared resource, others are stalled waiting for that resource. A context switch or task migration can alleviate the impact, but this behavior introduces unexpected variations in the cost of tasks (e.g. time for completion).

Memory, the most commonly shared resource, acts as an efficient method of communication between PEs. Several techniques have been employed to improve the way memory resources are shared, such as multiple memory banks to allow simultaneous access to variables in different banks and caches to reduce the number of memory requests. It is common that both memory banks and caches are shared by several PEs to reduce the complexity of the architecture at the price of slightly reducing the benefit.

Interconnects provide the mechanism for accessing some of these resources. According to their type and size, interconnects have complex arbitration rules that control the use of these resources. Shared memory and its interconnection with the PEs is normally the biggest source of uncertainty with respect to a task's cost. The growing number of PEs has made these structures very complex and their modeling involves stochastic processes [7, 8], making the cost of a task a random variable that is also a function, among other things, of other parameters in the architecture.

Statistics about the average costs and variances of tasks can be found to adjust the parameters of the SS. However, there are several aspects that limit the effectiveness of this approach: 1) The number of tasks is limited, so the disparities made by the variations in cost may not be overcome. These limitations may be due to available resources of the problem itself. 2) The load on each port of the crossbar or memory bank may vary, particularly when the size of a block of memory is not a multiple of the memory line size. A critical case would be where one memory bank is accessed by all processors. 3) The scenario becomes even more complex when we model the process as non-stationary, taking into account that memory transaction patterns can change in the same application over time.

```
// Globals
int TotalNumTasks;
int TaskIndex;
// Scheduling Function
int GetNewTask(int* Index){
  return (atomic_add(&Index, 1));
  }
```

```
// Scheduler algorithm on each PE
  int i;
  i = GetNewTask(&TaskIndex);
  while ( i<TotalNumTasks ){
    Execute_Task(i);
    i = GetNewTask(&TaskIndex);
  }
```

Figure 3: Code Fragment for a DS implementation

In the end, the simple model for SS is just an approximation that works very well for scenarios where the demand for shared resources is low because the stochastic component is negligible. For high performance scenarios, where these resources are required to be used at full capacity, congestion and arbitration require a more accurate model. However, a highly accurate model that considers a significant number of the variables that describe the behavior of the architecture, including the interactions of shared resources and their arbitration, is impractical for SS given the difficulty of producing such a model, the overhead of using the model to compute an optimal partition and the intrinsic limitations of the model.

Alternatively, DS performs the distribution of tasks at runtime allowing it to deal with all possible variations. With a competitive overhead, DS will be able to deliver better performance than SS, even on highly regular applications running under the new conditions previously described.

4.3 Low Overhead Fine grained Dynamic Scheduling

Dynamic Scheduling has been explored extensively for Instruction Level Parallelism and its advantages are well known [9, 10]. However, its implementation has traditionally required special hardware support.

For Task Level Parallelism, software implementations are preferred. When all tasks to be executed are similar and the parallel architecture is homogeneous, SS has been the preferred choice because the overhead for scheduling is only paid once and is largely independent of the data size whereas the overhead of dynamic scheduling grows linearly with the data size.

The overhead of DS can negate any advantages over SS if the implementation is not well designed. Unfortunately, special hardware support at the functional unit level is also impractical for general purpose many-core architectures.

Dynamically scheduling multiple, similar tasks, can be achieved with a single integer variable that is sufficient to uniquely identify a task in its set. A piece of code that implements Dynamic Scheduling in this manner is showed in figure 3.

The bottleneck of this algorithm is the function `GetNewTask(.)`. Specifically, it is the serialized access over the variable `TaskIndex`. We will use the throughput μ (maximum number of requests that can be serviced per unit of time) over the variable `TaskIndex` to determine the tradeoffs between task granularity and number of PEs. The lower bound for the size of a task can be obtained by considering that during the execution of a task, on average, all other

$(P-1)$ processors will request one task. Since the duration of the execution of a task T is given by $f(T)$, the lower bound for the average size of a task is

$$f(T) \geq \frac{P-1}{\mu} \tag{1}$$

Equation 1 shows that, as the number of PEs increase, a matching increase in throughput is required to guarantee scalability. Also, fine granularity of optimized tasks on a many-core environment requires the highest maximum throughput for the variable `TaskIndex` in order to avoid contention and lost performance under DS.

Unfortunately, implementations of the function `GetNewTask` that use locks or "inquire-then-update" approaches have very low throughput [11]. The main reason is that, for a lock implementation, the algorithm will 1) obtain a lock, 2) read and update the variable `TaskIndex` and 3) release the lock. A lock-based implementation of `GetNewTask` needs at least two complete roundtrips to memory, limiting its throughput to $\mu = \frac{1}{2q}$ where q is the minimum latency for a memory operation. Similarly, an "inquire-then-update" approach, such as Compare and Swap (CAS), requires `TaskIndex` to remain unchanged for at least 2 memory roundtrips, resulting in the same throughput as in the previous case.

We propose taking advantage of the support provided by *in-memory* atomic operations. In this case, each memory controller has an ALU that allows it to execute atomic operations directly inside the memory controller, without help from a thread unit, avoiding unnecessary roundtrip delays. In this case, the use of the *in-memory* atomic addition allows the throughput to be limited only by the time k taken by the memory controller to execute the operation, resulting in a throughput of

$$\mu = \frac{1}{k} \tag{2}$$

Cyclops-64 (C64) is an example of a many-core architecture that provides adequate hardware support for Dynamic Scheduling. In C64, implementations that use *in-memory* atomic additions enjoy a significant throughput increase because atomic operations in C64's memory controller take $k = 3$ cycles, whereas a roundtrip to on chip shared memory requires $q = 30$ cycles. It is a theoretical improvement of $20X$ over the throughput obtained using simple software implementations. In practice, the throughput will be lower because the memory controller is shared with the actual computation of the tasks. Nevertheless, this high throughput allows the overhead of DS to remain competitive the traditional SS approach.

5 Experimental Evaluation

In this section, we analyze the advantages of DS over SS for very regular workloads under the presence of shared resources and hundreds of PEs. We have illustrated different scenarios with fine grain tasks in order to compare the traditional SS and a low overhead DS. Our results

show that applications with many similar tasks scale better, and can take advantage of a low overhead DS, when the PEs are sharing resources and the amount of task is limited.

5.1 Experimental Testbed

We have chosen C64, a many-core processor architecture as the testbed architecture because it has a large number of processors sharing many diverse resources including, but not limited to, an on-chip memory, a crossbar switch and shared FPUs. In addition, it supports in-memory atomic addition, an essential component for a low overhead DS implementation as described in section 4.3.

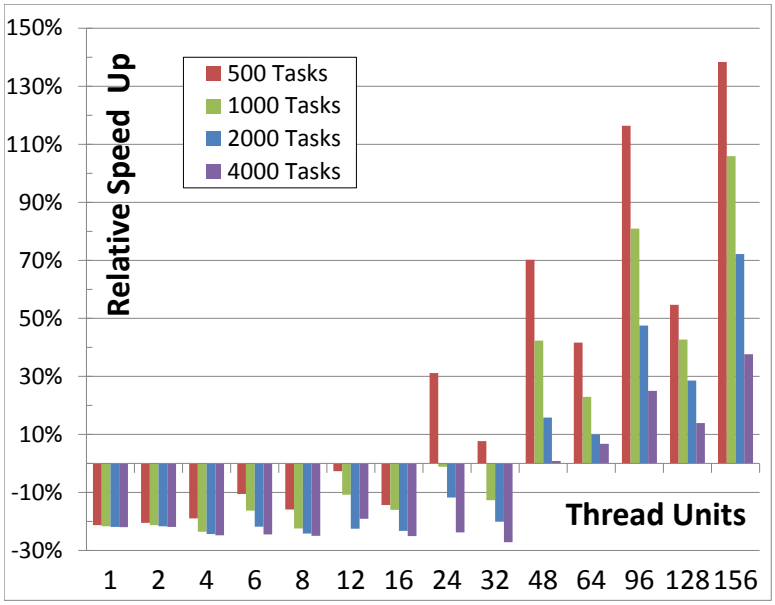
Our experiments were compiled with ET International's C64 C compiler, version 4.3.2, with compilation flag -O3. C64 processor chips are, as of the writing of this paper, available only to the US Government. For that reason, we ran our experiments with FAST [12], a very accurate simulator that has been shown to produce results that are within 10% of those produced by the real hardware. The simulator includes all the behaviors related to the arbitration of shared resources, as described in section 2.

We ran three different tests. The first is a microbenchmark that performs a memory copy of a vector in shared memory and computes a checksum on the elements of the vector. The second is a highly optimized Dense Matrix Multiplication using both on-chip and off-chip memory. The third test is a Sparse Vector Matrix Multiplication with variable parameters such as sparsity and variance of number of elements between columns. All benchmarks were implemented with an SS strategy that distributes work uniformly and a low-overhead DS that uses in-memory atomic addition.

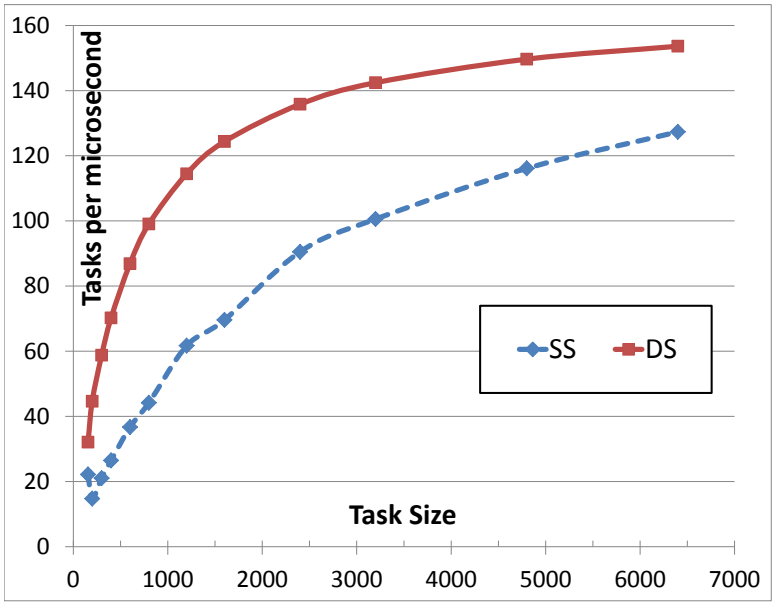
5.2 Memory Copy microbenchmark

The tasks in this microbechmark process 256 bytes of data from on-chip memory as follows: First, the PE copies a chunk of data from on-chip memory to local memory. Then it computes the checksum of the bytes and the chunk is stored back to another location in on-chip memory. Note that all tasks perform the same amount of work, but the arbitration of shared resources, like the crossbar switch, can result in varying performance as described in Section 4.2. We report the relative speed up of the DS approach with respect to its SS counterpart using the same number of PEs (Thread Units). We use different numbers of tasks to study and compare the behavior of SS and DS.

Figure 4(a) clearly shows the trade offs between DS and SS with respect to the number of PEs. As expected, when the number of PEs is small, DS cannot compete with SS and demonstrates a slowdown, with the worst case being 27% for 32 PEs. After 24 PEs, a smaller number of tasks start to give better performance with DS and after 48 PEs all the datasets favor DS. The maximum relative speed up is 137%, which is reached with the smallest problem size and the maximum number of PEs.



(a) Relative Speed Up of DS vs. SS



(b) Scalability for 156 PEs

Figure 4: Speed Up and Scalability of Memory Copy

We further studied the scalability of SS and DS with the maximum number of PEs allowed. Figure 4(b) shows the number of tasks per microsecond using different numbers of tasks. We note how DS scales better for cases with a limited number of tasks, quickly reaching the limit

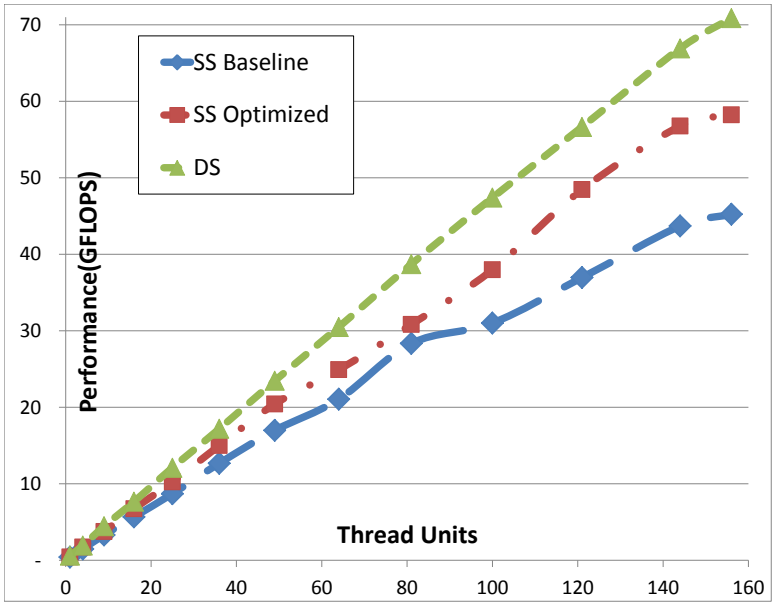


Figure 5: Performance for a DMM of size 486×486

imposed by crossbar congestion. At the same time, SS shows poor performance when compared to DS for every case.

5.3 Dense Matrix Multiplication

Dense Matrix Multiplication (DMM) exemplifies the type of highly regular and embarrassingly parallel application where SS seems to be the better choice over DS. We use, for our baseline, the Highly Optimized DMM for C64 using on-chip memory as described in Section 2 and detailed in [1]. We further increased the performance to 58.95 GFLOPS by using other static techniques, such as Data and Instruction Prefetching. Based on the observations made in Section 4, we implemented a DS for DMM using the same optimized register tiling described in [1]. With the implementation of DS, the maximum performance and scalability with respect to the number of PEs (Thread Units) increased significantly (figure 5).

The maximum performance reached is 70.87 GFLOPS, which is 88.86% of the theoretical peak performance. It is important to note the highly linear scalability with the number of PEs whereas the SS implementations start to show problems after only a hundred PEs. We further studied the scalability with respect to the matrix sizes. Figure 6(a) shows that the performance of DS increases significantly for smaller sized problems, with near maximum performance being reached using matrices of sizes 200×200. Note that the optimized SS version would be able to reach a slightly better performance than the DS version, given a suitably large problem size because of the constant overhead of SS. However, on-chip memory places an upper bound on the problem size making DS preferable for all implementations that use on-chip memory only.

We also studied the impact of the scheduling with larger matrices using off-chip memory. Because C64 has a software managed memory hierarchy, the programmer is in charge of the data movement between off-chip and on-chip memory. In order to sustain the performance reached in on-chip memory, overlapping of computation and data movement was used by implementing a double buffering schema. We determined, experimentally, that 8 PEs dedicated to data movement was enough to keep the remaining PEs working on computation.

Two versions of the DMM were implemented. In the static version, all tasks (computation and data movement) were determined and assigned statically from the beginning of execution. The necessary synchronization between tasks was performed using the low latency hardware barriers available on C64. In the dynamically scheduled version, tasks are available after satisfying their dependencies in a dataflow inspired manner [13] with a Dynamic Scheduler that takes advantage of the in-memory atomic operations available in C64.

The results in figure 6(b) show the high scalability and excellent performance reached by the Dynamic Scheduling implementation, whereas the Static version is not able to surpass half the theoretical peak performance of C64. Furthermore, the scalability of the SS implementation decreases after 120 PEs.

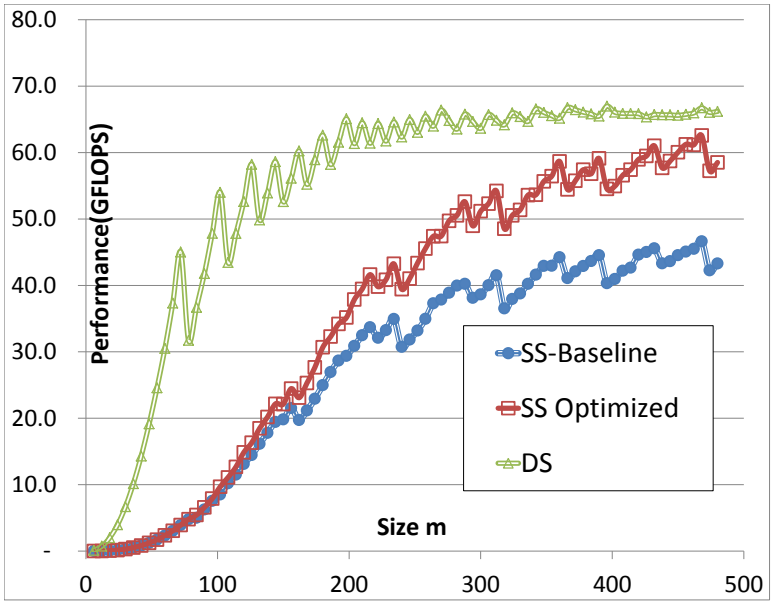
5.4 Sparse Vector Matrix Multiplication

Sparse linear algebra applications present additional challenges to their dense counterparts, including variable memory access patterns and other difficulties related to the particular structure of the sparse matrices. We use the Sparse Vector Matrix Multiplication (SpVMM) defined by equation 3 where A is a sparse matrix of size $m \times n$, and v and w are vectors of lengths m and n respectively.

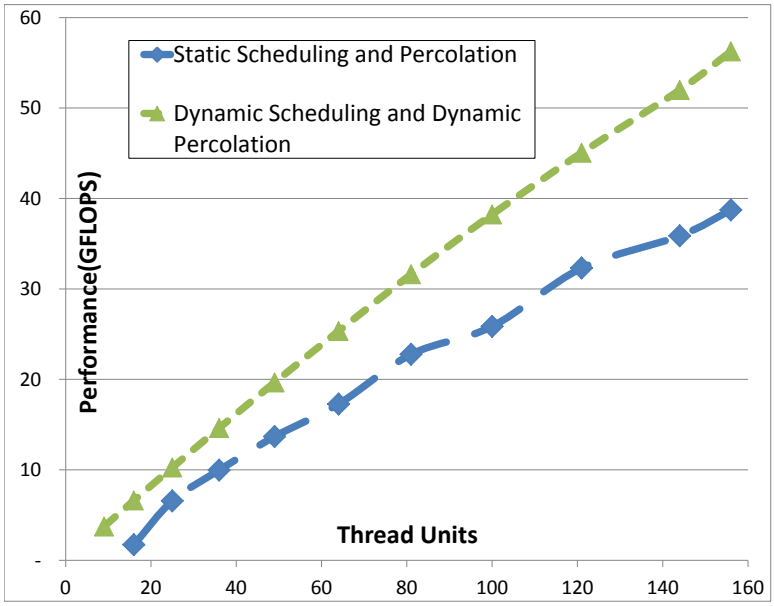
$$w_j = \sum_{A_{i,j} \neq 0} v_i A_{i,j} \tag{3}$$

The sparse matrix A is stored using the Compressed Spare Column format (CSC). A task is defined as the computation of one element of w. Two parameters were varied to explore different behaviors of the SpVMM: The sparsity s varies in the range $[0 - 1]$ and defines the number of non-zero elements (NNZ). The non-zero elements are distributed uniformly across columns with a normalized variance u in the range $[0 - 1]$. The matrix is generated randomly without any particular spatial locality in order to maintain a balanced distribution of tasks among PEs both for SS and DS implementations. Task sizes have slight differences according to the normalized variance u.

Figure 7 shows the relative speed up of DS with respect to its SS counterpart with the same characteristics. (including number of PEs). All the matrices have $n = 400$. The results are reported for different sizes of the tasks m and sparsity of the matrix s. In addition, the experiments were made using 3 possible values of the normalized variance $u = \{0.1, 0.5, 0.9\}$. The results illustrate how, with a high variance of task sizes, DS overcomes SS even with few



(a) Scalability for a DMM with 144 PEs



(b) Scalability for a DMM of size 6480×6480

Figure 6: Scalability of Dense Matrix Multiply

PEs. If the variance between tasks is decreased, SS has better performance than DS when the number of PEs is small but SS cannot scale properly when the number of PEs increases. Even in the case of very similar tasks ($u = 0.1$), DS has higher performance than SS for 128 PEs.

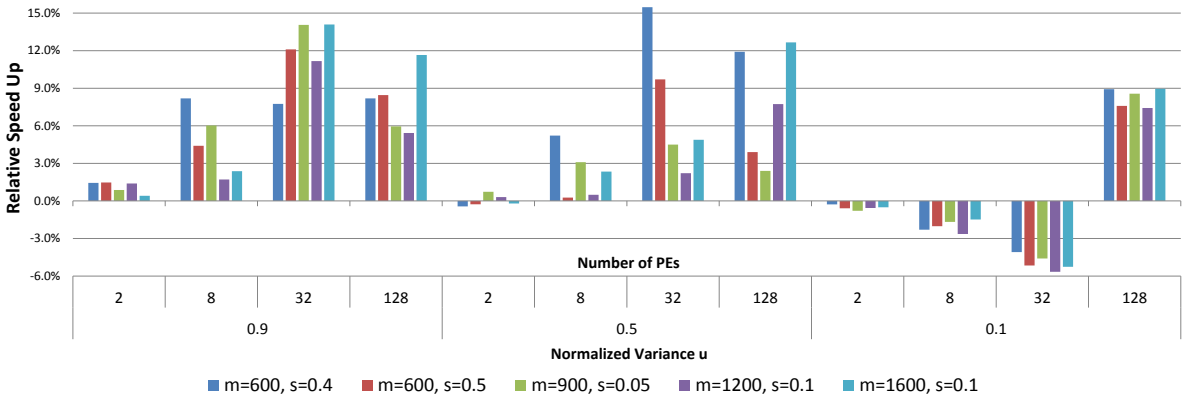


Figure 7: Relative Speed Up of DS vs. SS for SpVMM

6 Related Work

The Intel Larabee many-core architecture implements task scheduling entirely with software, allowing for a lightweight DS [14]. The project has evolved into the Many Integrated Core (MIC) and there are few details about the performance of the scheduler. However, this work provides evidence indicating the importance and complexity of task scheduling on many-cores, even under homogeneous workloads.

In general, runtime implementations are focused on scheduling loads that are frequently heterogeneous, based on queues and focused on locality. One of the most popular examples is Cilk [15]. Other approaches include EARTH [16] and Habanero [17]. Our contribution is more focused on explaining the challenges of scheduling fine-grained homogeneous tasks on many-core architectures and, in the process, showing some limitations of SS with regard to scalability and shared resources, and how those limitations are overcome by DS.

Several approaches consider the task scheduling problem as a Bin-Packing problem. Different scheduling techniques have been proposed according to the desired optimization function. Good summaries can be found in [18, 19]. Most are not architecture aware and do not consider the overhead and arbitration of shared resources, which is especially important for finer granularity. They have been useful for coarse grained tasks and distributed systems providing boundaries for optimum scheduling strategies.

7 Conclusions and Future Work

We have shown that, for highly regular and embarrassingly parallel applications, DS is preferable over SS in scenarios commonly found in many-core architectures. These scenarios involve the presence of shared resources under different arbitration policies, hundreds of processing units, and a limited amount of work.

We explained how these factors degrade the expected performance of SS and how DS behaves better under these conditions. The presence of shared resources, such as a crossbar switch, produces unexpected and stochastic variations on the duration of tasks that SS is unable to manage. In addition, a uniform mapping of work to processors without considering the granularity of the tasks is not necessarily scalable under limited amounts of work.

We also explained how the advantages of DS are enhanced by a low-overhead implementation using mechanisms provided by the architecture, particularly *in-memory* atomic operations, diminishing the overall overhead of DS. As a result, finer task granularities can make the use of DS efficient.

These factors allow DS to scale better than SS as the number of processors increase. We demonstrated how Dynamic Scheduling can overcome Static Scheduling with regard to performance. We did this with a synthetic microbenchmark and two applications. Particularly, DMM was able to reach 70.87 GFLOPS out of 80 GFLOPS.

Our future work will be focused on the development of an accurate performance model for DS involving certain parameters of the architecture and the characteristics of the tasks. We also plan to extend the analysis and implementation made for homogeneous tasks to a more general case. We are especially interested in how in-memory atomic operations can be used in the design of high throughput queues.

References

[1] E. Garcia, I. E. Venetis, R. Khan, and G. Gao, "Optimized dense matrix multiplication on a many-core architecture," in *Euro-Par 2010*, Ischia, Italy, 2010.

[2] L. Chen, Z. Hu, J. Lin, and G. R. Gao, "Optimizing the Fast Fourier Transform on a Multi-core Architecture," in *IEEE IPDPS '07*, Mar. 2007, pp. 1–8.

[3] I. E. Venetis and G. R. Gao, "Mapping the LU Decomposition on a Many-Core Architecture: Challenges and Solutions," in *CF '09*, Ischia, Italy, May 2009, pp. 71–80.

[4] D. A. Orozco and G. R. Gao, "Mapping the fdtd application to many-core chip architectures," in *ICPP '09*. Washington, DC, USA: IEEE, 2009, pp. 309–316.

[5] M. Denneau and H. S. Warren Jr., "64-bit Cyclops: Principles of Operation," IBM Watson Research Center, Yorktown Heights, NY, Tech. Rep., April 2005.

[6] Z. Hu, J. del Cuvillo, W. Zhu, and G. R. Gao, "Optimization of Dense Matrix Multiplication on IBM Cyclops-64: Challenges and Experiences," in *Euro-Par 2006*, Dresden, Germany, Aug. 2006, pp. 134–144.

[7] I. Kaj, *Stochastic Modeling in Broadband Communications Systems*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2002.

[8] M. Lin and N. McKeown, "The throughput of a buffered crossbar switch," *Communications Letters, IEEE*, vol. 9, no. 5, pp. 465 – 467, may 2005.

[9] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM J. Res. Dev.*, vol. 11, pp. 25–33, January 1967.

[10] J. Stark, M. D. Brown, and Y. N. Patt, "On pipelining dynamic instruction scheduling logic," *Microarchitecture, IEEE/ACM International Symposium on*, vol. 0, p. 57, 2000.

[11] D. Orozco, E. Garcia, R. Khan, K. Livingston, and G. Gao, "High throughput queue algorithms," *CAPSL Technical Memo 103*, January 2011.

[12] J. del Cuvillo, W. Zhu, Z. Hu, and G. R. Gao, "FAST: A Functionally Accurate Simulation Toolset for the Cyclops-64 Cellular Architecture," in *MoBS '05*, 2005, pp. 11–20.

[13] E. Garcia, R. Khan, K. Livingston, I. Venetis, and G. Gao, "Dynamic percolation - mapping dense matrix multiplication on a many-core architecture," *CAPSL Technical Memo 098*, June 2010.

[14] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: a many-core x86 architecture for visual computing," *ACM Trans. Graph.*, vol. 27, pp. 18:1–18:15, August 2008.

[15] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: an efficient multithreaded runtime system," *SIGPLAN Not.*, vol. 30, pp. 207–216, August 1995.

[16] K. B. Theobald, "Earth: an efficient architecture for running threads," Ph.D. dissertation, Montreal, Quebec, Canada, 1999.

[17] Y. Guo, R. Barik, R. Raman, and V. Sarkar, "Work-first and help-first scheduling policies for async-finish task parallelism," in *IPDPS'09*. Washington, USA: IEEE, 2009, pp. 1–12.

[18] E. G. Coffman, *Bounds on Performance of Scheduling Algorithms. In Computer and Job Shop Scheduling Theory*. New York: Wiley, 1976.

[19] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson, *Approximation algorithms for bin packing: a survey*. Boston, MA, USA: PWS Publishing Co., 1997, pp. 46–93.