A Clustered Approach to Multithreaded Processors[1]

Venkata Krishnan and **Josep Torrellas**

Department of Computer Science
University of Illinois at Urbana-Champaign, IL 61801
venkat,torrella@cs.uiuc.edu
http://iacoma.cs.uiuc.edu/iacoma/

Abstract

With aggressive superscalar processors delivering diminishing returns, alternate designs that make good use of the increasing chip densities are actively being explored. One such approach is *simultaneous multithreading* (SMT), where a conventional superscalar supports multiple threads such that instructions from different threads may be issued in a single cycle. Another approach is the on-chip multiprocessor and its variants. Unlike the SMT approach, all the resources have *fixed assignment* (FA) in this architecture. The design simplicity of the FA approach enables high clock frequencies, while the flexibility of the SMT approach allows it to adapt to the specific thread- and instruction-level parallelism of the application. Unfortunately, the strict partitioning of resources among various processors in the FA architecture may result in underutilization of the chip, while the fully centralized structure of the SMT may result in a longer clock cycle-time.

In this paper, we explore a hybrid design, where a chip is composed of a set of SMT processors. We evaluate such a clustered architecture running parallel applications. We consider both a low-end machine with only one processor chip on which to run multiple threads as well as a high-end machine with several processor chips working on the same application. Overall, we conclude that such a hybrid processor represents a good performance-complexity design point.

1 Introduction

Today's high-performance superscalar processors such as the PowerPC, Pentium-Pro or R10000 issue multiple instructions per cycle. The number of instructions issued is limited by the dependences among instructions and the resources available in the processor. To uncover much instruction-level parallelism (ILP), these processors maintain a large window of instructions from which instructions are issued based on the availability of operands and resources. Unfortunately, these processors are becoming very complicated in their quest to extract more ILP.

A natural solution to compensate for the lack of ILP in a single thread is to divide the application into multiple control flows or threads and exploit ILP across them. An application can be divided into multiple threads by the compiler [1, 5] or by user hand-parallelization. In addition, several proposed software and hardware features can enable even sequential applications to execute in multithreaded mode [3, 4, 7, 13].

Once we have multiple threads, different architectures can be used. A promising approach is the on-chip multiprocessor [11] and its variants such as the Multiscalar [14], and the Superthreaded architectures [15]. In this approach, each processor in the chip handles one thread and exploits ILP within that thread. The main characteristic is that a processing unit in a chip is allocated exclusively to a thread with all resources having a fixed assignment to the thread. We refer to this type of architecture as the *fixed assignment* (FA) architecture. A major advantage of this approach is the simplicity of the design, which enables high clock frequencies. However, to fully utilize the chip, it is necessary to have as many threads as processors. The consequence is that resources such as fetch or functional units can be easily wasted when there is a lack of parallel threads or when a thread running on a processor cannot utilize the resource due to some hazard.

To utilize the resources better, we can design a more centralized architecture, where threads can share all the resources. This is the approach taken by Simultaneous Multithreading (SMT) [17]. In this scheme, the processor can support multiple threads such that, in a given cycle, instructions from different threads can be issued. The major advantage of this approach is that the resources tend to be highly utilized. If, in a given cycle, a thread is not using a resource, that resource can, typically, be utilized by another thread. Tullsen *et al* [16] describe a fully centralized SMT architecture with a relatively small impact on a conventional superscalar design. Evaluation of this architecture when running multiprogrammed and parallel workloads [6, 9, 16] has shown significant speedups. However, a drawback of this approach is that it may inherit all the complexity of existing superscalars and, in addition, add extra hardware. In fact, with the delays in the register bypass network dictating the cycle-time of future high-issue processors [12], a fully centralized SMT processor will likely have a slower clock frequency.

A very intuitive alternative to these two architectures is a hybrid of the FA and the centralized SMT approaches, namely a clustered SMT architecture. In this architecture, the chip has several independent processing units, with each unit having the capability to perform simultaneous multithreading. Given that the effort to enhance a conventional dynamic superscalar to perform simultaneous multithreading is small [16], a low-issue SMT processor is feasible. A clustered SMT architecture would be a simple design as it would involve replicating several such low-issue SMT processors on a die. However, it might be argued that this separation of processing units would restrict the sharing of resources. But, as we will show in this paper, this architecture is able to extract most of the benefits that can be achieved with the fully centralized SMT approach.

In this paper, we explore such a clustered SMT architecture when running parallel applications. We show the performance of both a low-end machine with only one processor chip to run the multiple threads, as well as a high-end machine with several processor chips working on the same application. The clustered SMT architecture often delivers a performance close to that of a fully dynamic centralized SMT. Even this limited form of simultaneous multithreading results in high performance due to the ability to adapt to the specific thread- and instruction-level parallelism of the application. Consequently, the hybrid architecture represents a good performance-complexity design point.

[1] This work was supported in part by the National Science Foundation under grants NSF Young Investigator Award MIP-9457436, ASC-9612099 and MIP-9619351, DARPA Contract DABT63-95-C-0097, NASA Contract NAG-1-613, and gifts from IBM and Intel.

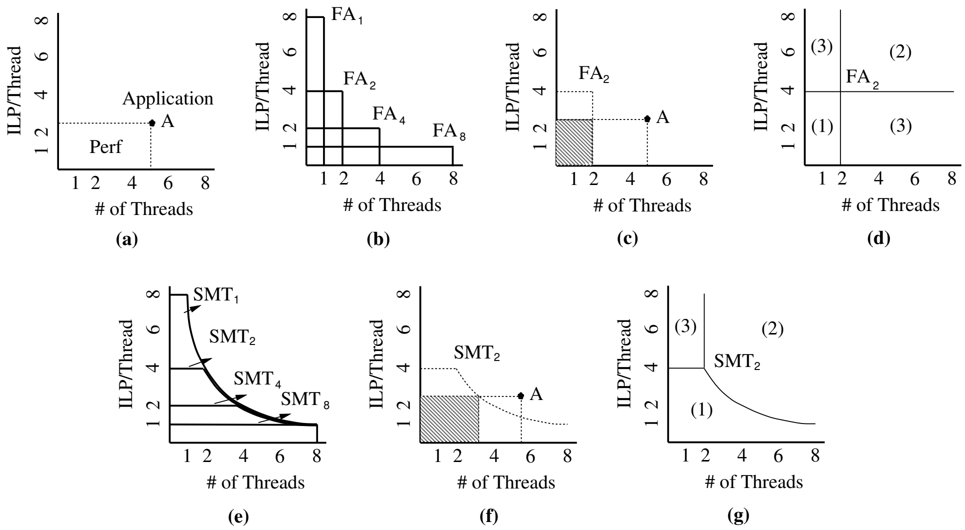


Figure 1: Model of parallelism.

This paper is organized as follows: Section 2 presents a model of parallelism that relates instruction- and thread-level parallelism; Section 3 describes the architectures analyzed; Section 4 discusses the evaluation methodology; Section 5 evaluates the architectures under several environments; and finally, Section 6 concludes the paper.

2 A Model of Parallelism

Before performing any architectural analysis, we propose a simple model of parallelism to gain insight into the performance of applications on processors that exploit both thread- and instruction-level parallelism. This model is simple and does not take into account any cycle-time variations that may occur between different designs. It just serves to illustrate the potential of each architecture. The model uses a chart that graphs the number of threads versus the ILP per thread (Figure 1-(a)). If we consider average conditions for simplicity, a given application A will fall somewhere in this chart. The area of the rectangle formed by the axes and the X and Y coordinates of A is proportional to the performance that can be extracted from this application.

Consider now the chart from a processor architecture point of view. We restrict ourselves to 8-issue processors. The different fixed assignment architecture (FA) configurations are shown in Figure 1-(b). For example, a FA with 8 single-issue processors is shown as the FA_8 rectangle. The rectangle means that any application that falls inside it will be fully exploited by the processor. The area of the rectangle is proportional to the maximum performance that can be delivered by the processor. In the figure, we show boxes for a FA processor with four 2-issue processors (FA_4), two 4-issue processors (FA_2), and a conventional 8-issue superscalar (FA_1). Consider application A again (Figure 1-(c)). FA_2 is unable to fully exploit the application. The performance delivered is proportional to the shaded area only, not to the whole rectangle under vertex A. In addition, the processor is under-utilized. While it can deliver as much performance as the size of its box, it is only delivering the performance of the shaded area. Overall, we can distinguish 3 regions depending on the relative position of the architecture and application boxes (Figure 1-(d)). If the application falls into region (1), the application is fully exploited and the processor is under-utilized. This means that the maximum performance for that application is achieved. In region (2), the application is under-exploited and the processor is fully utilized. This means that the processor is functioning at its optimum. Finally, in region (3), the application is under-exploited and the processor is under-utilized. Since we want to utilize the processor completely, we want applications to fall in (2). We call region (2) the *optimal* region.

SMT processors and clustered SMT processors can be represented as rectangles with the lower left vertex in the (0,0) coordinate and the opposite vertex sliding along a hyperbola (Figure 1-(e)). We consider SMT processors that can handle up to 8 threads. Consider first a centralized SMT processor. This is simply an 8-issue 1-cluster SMT processor that can issue instructions from any of the 8 threads. We refer to it as SMT_1, where the subscript 1 stands for the number of clusters in the SMT processor. This processor has the flexibility to adapt to the application's demands and appear as a FA_1, FA_2, FA_4 or a FA_8 processor. Furthermore, it can also appear, for example, as a FA with an average of 5 processors and an average issue width of 1.6 per processor. Since the product of processors and issue width per processor is constant, we follow the hyperbola shown in the figure. The rectangle slides its upper right vertex along the hyperbola. The maximum performance that can be delivered is constant, since the area of the rectangle is constant, but the rectangle can move and *adapt to the demands of the application.* The other, more limited clustered SMT processors are also represented by the same moving rectangle. However, their rectangle cannot follow the hyperbola all the way. Instead, it stops at a certain (X, Y) coordinate on the hyperbola and cannot move to higher Y coordinates. For example, a clustered SMT with two 4-issue processors (SMT_2), where each cluster can handle 4 threads, stops at (2,4). This is because it cannot exploit ILP in a thread that is higher than 4. In the figure, we marked this limitation with a horizontal line at Y = 4.

Consider now application A running on the SMT_2 processor (Figure 1-(f)). The shaded area is the performance delivered. The shaded area is larger than the one in Figure 1-(c). Therefore, the clustered architecture extracts more performance from the same application than the FA processor. We distinguish 3 regions (Figure 1-(g)). If the application falls into (1), the application is fully exploited and the processor is under-utilized. In region (2), which is the optimal region, the application is under-exploited and the processor is fully utilized. In region (3), the application is under-exploited and the processor is under-utilized.

From this model, therefore, we can see that SMT and clus-

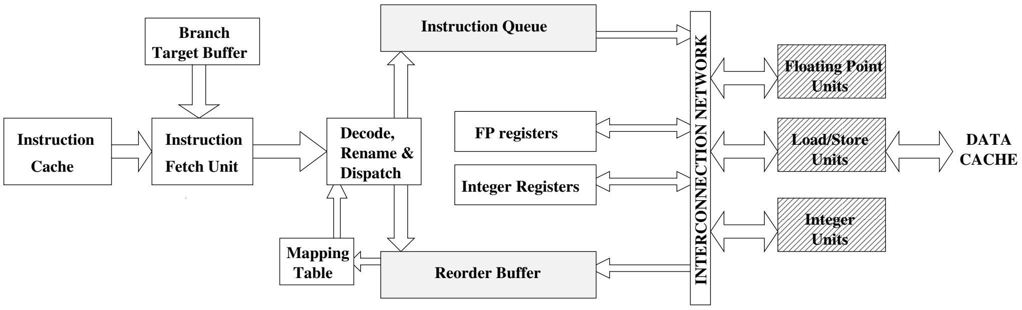


Figure 2: Architecture of the dynamic superscalar core used in all the architectures.

Integer Unit

Operation	Latency
add, sub, log	1
shift	1
mul	2
div	8
branch	1

Load/Store Unit

Operation	Latency
load	2
store	1

Floating Point Unit

Operation	Latency
fpadd	1
fpmult	2
fpdiv	4/7

Table 1: Characteristics of the functional units.

tered SMT processors are likely to deliver more performance than FA processors. The reason is that the optimal region for the former two processors is a superset of the optimal region for the FA processors.

3 Architectures Evaluated

In this section, we describe the different architectures evaluated.

3.1 Base Superscalar Core

We assume an aggressive dynamic superscalar core as the building block for all the architectures considered. This dynamic superscalar core can fetch and retire up to n instructions each cycle. It has a large fully associative instruction window along with integer and floating-point registers for renaming. The actual numbers are shown in Table 2. A 2K-entry direct-mapped branch prediction table, with each entry having a 2-bit saturating counter and addressed by the low-order bits of the PC, allows multiple branch predictions to be performed even when there are pending unresolved branches. Figure 2 gives a block diagram of the architecture.

Caches are non-blocking with up to 32 outstanding loads allowed with full load bypassing enabled. Table 1 gives details on the functional units and the instructions that they handle.

3.2 Centralized and Clustered SMT Architectures

The fully centralized SMT architecture is on the lines of [16]. It can support up to 8 threads and can issue up to 8 instructions per cycle (Table 2). The instruction fetch unit is shared by all threads, with each thread having a program counter. The fetch unit fetches instructions from a different thread every cycle in a round-robin fashion. In a given cycle, up to 8 instructions can be fetched. Instructions from different threads are held in a common 128-entry associative instruction window from where they may be issued in any order. Finally, instructions are committed on a per-thread basis. Although this is a very flexible architecture, resource centralization can make it quite complex to design. Specifically, one of the major bottlenecks in the architecture is the large interconnection network between the functional units and the register file. Part of this network, namely the bypass network, may actually determine the clock cycle-time of high-issue processors [12].

3.3 Fixed Assignment Architectures

To alleviate the bypass delay problem and also reduce the resource centralization, we partition the SMT processor into several clusters. For instance, SMT_2 has 2 clusters where each cluster is a SMT processor that supports 4 threads. Each cluster has its own fetch unit, with a thread capable of fetching up to 4 instructions/cycle. The ability to bypass across clusters (with an additional delay) could also be provided, though we do not assume one in this paper. Overall, unlike in the centralized scheme, no resource sharing is done across clusters. Therefore, with a single thread, this processor can achieve at most 4 IPC. Similarly, the SMT_4 processor has 4 clusters, each supporting 2 threads and able to issue 2 instructions per cycle (Table 2).

Note that the cycle-time of these clustered architectures is much smaller than that of the centralized SMT. Indeed, Palacharla and Jouppi [12] estimate that the cycle-time for an 8-issue processor will be twice as long as a 4-issue processor when using 0.18μ technology. In the light of their observations, SMT_2, with two 4-issue clusters, may have a frequency that is twice higher than SMT_1. However, in this paper, we ignore this effect and compare the performance assuming the same cycle-time.

We consider 3 variations of the FA architecture. In this architecture, each individual thread runs in its own cluster using its own set of resources. The first type (FA_2) is a processor with two 4-issue clusters. Each cluster handles a single thread, for a total of 2 threads per processor. The second type (FA_4) is a processor with four 2-issue clusters. Each cluster handles a single thread, for a total of 4 threads per processor. Finally, FA_8 is a processor with eight 1-issue clusters, for a total of 8 threads/processor. In terms of cycle-time and space requirements, SMT_2 is similar to FA_2, while SMT_4 is similar to FA_4. This is because, unlike in a 8-issue superscalar, the register bypass delay does not decide the cycle-time in a 4-issue superscalar [12].

Details on the configuration of the various processor archi-

Processor Type	Number of Clusters, Max. IPC/cluster	Threads per Cluster [Chip]	Number of Functional Units (int/ld-st/fp) per Cluster [Chip]	Entries in Instruction Queue & Reorder buffer per Cluster [Chip]	Number of Renaming Registers (int/fp) per Cluster [Chip]
FA_8	8, 1	1 [8]	1/1/1 [8/8/8]	16 [128]	16/16 [128/128]
FA_4	4, 2	1 [4]	2/2/2 [8/8/8]	32 [128]	32/32 [128/128]
FA_2	2, 4	1 [2]	4/4/4 [8/8/8]	64 [128]	64/64 [128/128]
FA_1	1, 8	1 [1]	6/4/4 [6/4/4]	128 [128]	128/128 [128/128]
SMT_4	4, 2	2 [8]	2/2/2 [8/8/8]	32 [128]	32/32 [128/128]
SMT_2	2, 4	4 [8]	4/4/4 [8/8/8]	64 [128]	64/64 [128/128]
SMT_1	1, 8	8 [8]	6/4/4 [6/4/4]	128 [128]	128/128 [128/128]

Table 2: Description of the different types of architectures evaluated.

Parameter	Value
[L1 / L2] cache size (Kbytes)	[64 / 1024]
[L1 / L2] cache line size (Bytes)	[64 / 64]
[L1 / L2] cache associativity	[2-way / 4-way]
[L1 / L2] cache fill time (Cycles)	[8 / 8]
Number of banks in [L1 / L2] cache	[7 / 7]
[L1/ L2] cache Read or Write occupancy	[1 / 1]
L1 latency (Cycles)	1
L2 latency (Cycles)	10
Local memory latency (Cycles)	40
Remote memory latency (Cycles)	60
Remote L2 latency (Cycles)	75

Table 3: Characteristics of the memory hierarchy. All latencies correspond to a contention-free round trip.

tectures are given in Table 2.

3.4 Memory Subsystem & Multiprocessor Configuration

Typically, each cluster in a processor would have its own private primary cache and share the secondary cache. In our work, however, we wanted to avoid the results being influenced by different memory hierarchies in different processors. Consequently, we choose a shared primary cache for all our configurations. The characteristics of the cache hierarchies are shown in Table 3.

In our simulations, we model contention in great detail. The no-contention latency of round-trip accesses to different levels of the memory hierarchy is shown in Table 3. The remote memory and remote cache latencies are applicable only when we simulate a machine with multiple processor chips. The instruction misses in the parallel applications used in this paper are minimal. Consequently, we assume a perfect instruction cache. Finally, the 512-entry TLB is shared by all threads and is fully associative and uses random replacement.

For the low-end machine, we model a simple workstation. For the high-end machine, we model a scalable shared-memory multiprocessor similar to DASH [8] as shown in Figure 3. Each node has a portion of the global shared memory and directory. The remote latencies, specified in Table 3, are low because we only model a 4-node machine.

4 Evaluation Environment

Our simulation environment is built upon the MINT [18] execution-driven simulation environment. MINT captures both application and library code execution and generates events by instrumenting binaries. We have modified the MINT front-end to handle MIPS2 binaries as well as instrument basic block boundaries. The application is executed through MINT, which generates basic block and memory events. This information is used by our back-end simulator. Our processor simulator is extremely detailed and performs a cycle-accurate simulation of all the architectures described in Table 2.

For a parallel application, we generate as many threads as are required by the processor. Thus, the application runs in sequential mode when executing on a conventional superscalar (FA_1) where only one thread is created. For FA_8 as well as for SMT processors, 8 threads are created.

We choose six applications. Three applications are SPEC95 benchmarks, namely, *swim*, *tomcatv* and *mgrid*; one is a NASA7 kernel, *vpenta*, and the remaining two are SPLASH-2 applications [19], namely *fmm* and *ocean*. The two SPLASH-2 applications are explicitly-parallel programs written in C and use ANL m4 macros [10] for parallel constructs. The SPEC95 benchmarks and the NASA7 kernel are sequential programs written in Fortran. We use the Polaris automatic parallelizing compiler [1] to identify parallel sections of the code. Polaris uses several techniques, such as inter-procedural symbolic program analysis, scalar and array privatization, symbolic dependence analysis, advanced induction and reduction variable recognition and elimination, to parallelize a Fortran application. In these Fortran applications, little or no programmer modification of the sequential code is required. The default data set was used for all applications. However, the number of time steps was reduced to let the simulations complete faster. The statistics include both serial as well as parallel sections of the applications.

4.1 Statistics Collection

We gather detailed statistics on an issue slot basis. For each processor, we scan the entire instruction window every cycle and record the type of hazard faced by each instruction that is unable to issue. At the end, the wasted slots are divided proportionally among the different types of hazards. The different categories of hazards are: lack of functional units (*structural*), memory access (*memory*), data dependencies (*data*), branch mispredictions (*control*), spinning on barriers or locks (*sync*) and no instructions for a thread in the instruction window (*fetch*). There is also an *other* category for slots wasted due to instructions squashed on a branch misprediction or when there is a stall due to lack of renaming registers. Fi-

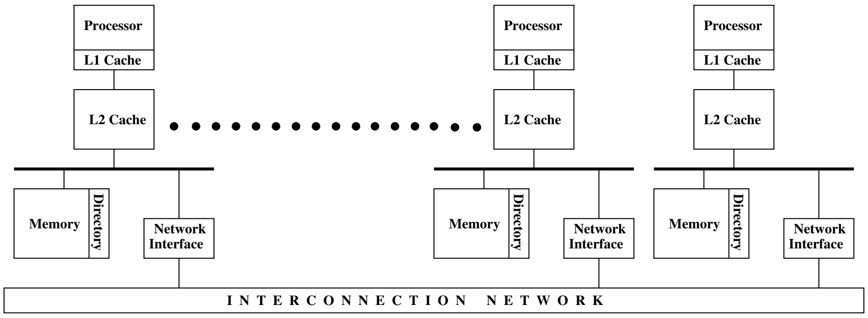


Figure 3: Multiprocessor configuration.

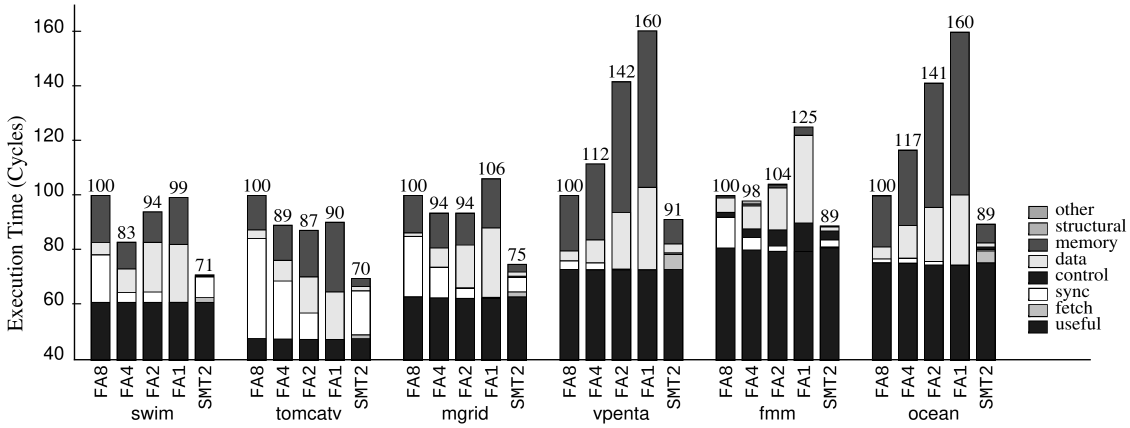


Figure 4: Comparing FA processors to a clustered SMT for a low-end machine. The chart assumes the same cycle-time for all the architectures.

nally, the useful instruction slots are grouped under *useful*.

5 Evaluation

We now evaluate the performance of the clustered SMT, FA, and centralized SMT architectures. In our evaluation, we model two environments, namely a low- and high-end machine. In the former, there is only one processor chip to run the multiple threads, while in the latter, there are several processor chips working on the same application, each running the multiple threads. Section 5.1 considers the clustered SMT and the FA architectures, while Section 5.2 considers the clustered and centralized SMTs.

5.1 Clustered SMT and FA Architectures

Figure 4 shows the execution time of the applications on the FA processors and the SMT_2 clustered processor. We have selected SMT_2 as the representative for the clustered SMT architectures. The reason for this choice will become obvious in the next section. Figure 4 corresponds to a low-end machine environment. For each application, the bars are normalized to those of FA_8. Recall that, in all the charts, we measure the execution time in number of cycles. We will take into account the actual cycle time later.

We start by comparing the FA processors. As the number of processors per chip decreases, the contribution of the *sync* hazard steadily decreases, while the *data* and *memory* hazards steadily increase. The contribution of *sync* decreases for two reasons. First, with fewer threads, the relative impact of the serial section of the code or the load imbalance decreases. Secondly, the thread that executes the serial section can speed it up more because it can exploit more ILP. Unfortunately, the benefits of the higher issue per thread provide diminishing returns. Soon, *data* and *memory* hazards appear.

In each application, the bars often take a U-shape as we increase the issue width of the processors in the FA processors from one (in FA_8) to eight (in FA_1). The configuration with the lowest execution time or *sweet-spot* depends on the application. When the application is highly parallelizable, there is little cost involved with using many threads. In this case, FA_8 has the minimum execution time. *Vpenta* and *ocean* belong to this class of applications. However, when the application has low levels of parallelism, the benefits of exploiting parallelism across threads are not as high as those of exploiting ILP. Therefore, fewer wider-issue processors will perform better. For example, the FA_4 is the best for *swim* and *fmm*, while FA_2 is the best for *tomcatv* and *mgrid*. Overall, therefore, no FA processor is clearly the best.

We now consider the clustered SMT_2 processor. From the figure, we see that it takes the fewest cycles for all the applications. In addition, it delivers the most stable performance: on an average, it takes 13% lesser execution cycles than the best FA processor for a particular application. The main advantage of the SMT_2 processor is its ability to adapt to the thread parallelism and ILP requirements of the application. For applications with high thread parallelism like *ocean*, it performs better than FA_8, while for applications with low thread parallelism like *tomcatv*, it performs better than FA_2 and FA_1.

We now examine a high-end machine. We model a machine with four processor chips working on the same application. In such an environment, two major changes occur. The first one is that, because of Amdahl's law, sections that are serial or have load imbalance become more important. It is therefore

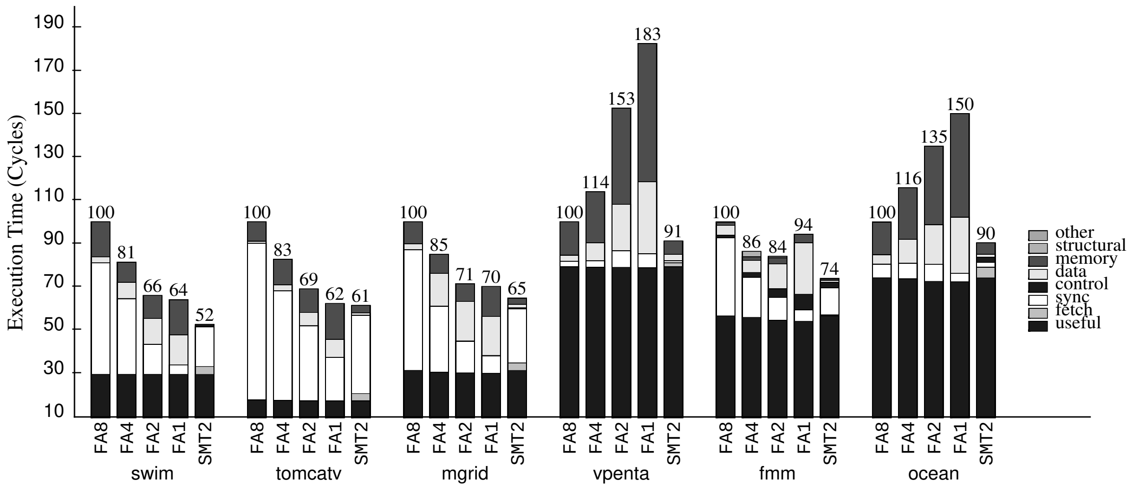


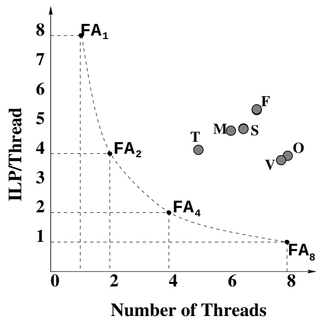
Figure 5: Comparing FA processors to a clustered SMT for a high-end machine. The chart assumes the same cycle-time for all the architectures.

crucial to speed them up. This argues for using a wide-issue processor. The second change is that the parallel sections suffer more hazards. This is because memory latencies are higher and synchronization occurs more frequently. This argues for SMT-based processors, where the issue cycles can be shared between threads. Overall, this should mean that SMTs are more attractive because, in the serial section, they can become wide-issue processors while, in the parallel section, they allow issue cycles to be shared between threads.

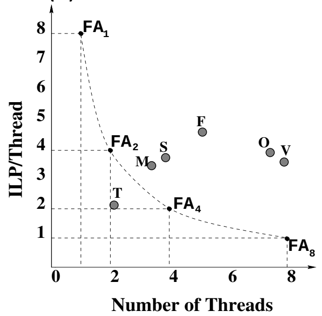
Figure 5 shows the execution time of a 4-processor high-end machine configured with the five types of architectures shown in Figure 4. Since there are 4 chips, FA_8 and SMT_2 run with 32 threads, while FA_4, FA_2 and FA_1 run with 16, 8 and 4 threads respectively. The figure is organized in the same way as Figure 4.

We focus first on the FA processors. For the least parallel applications, namely *swim*, *tomcatv* and *mgrid*, we see that the *sweet-spot* of their U-shape has moved to the wider issue FA processor. Indeed, while in Figure 4, FA_4 or FA_2 performed best, in Figure 5, FA_1 performs best. This is because it is now critical to speed up serial sections with wide-issue processors. This shows that, more than in a low-end system, exploiting ILP is very important in a high-end system. On the other hand, if we examine highly parallel applications with little synchronization like *vpenta*, often the opposite occurs. FA_1 becomes relatively worse than in Figure 4. This is because parallel sections become more hazardous. Overall, therefore, in a multi-chip environment more than in a single-chip one, no single FA processor performs best. The variations in performance among the FA processors have exacerbated. In contrast, the SMT_2 processor has the lowest execution time of all processors. In addition, it delivers the most stable performance by far. We conclude, therefore, that SMT_2 processors are even more attractive in high-end systems.

Recall that, for the above evaluation, we did not consider variations in clock cycle-time between different architectures. However, even if we considered them, the broad conclusion would not change. This is because the FA_2 architecture has largely the same cycle-time as SMT_2. The reason is that both architectures have 4-issue superscalar clusters. These 4-issue superscalars do not have as much cycle-time constraints as higher-issue processors [12]. In addition, with current technology [2], the difference in cycle-time between 4-issue processors and lower-issue processors (FA_4 and FA_8) is very unlikely to be significant enough to make up for the large difference in performance shown in the figure. Finally, FA_1, which is a conventional superscalar, is likely to have a larger cycle-time than FA_2 or SMT_2 [12].



(a) Low-end machine.



(b) High-end machine.

Figure 6: ILP versus thread parallelism for our applications. Each application is denoted by the first letter in its name.

5.1.1 Comparing the Results to the Model

We now qualitatively compare the results obtained to the performance predicted by our model of parallelism in Section 2. For each application, we estimate the thread parallelism and the ILP roughly. We estimate the thread parallelism by measuring the average number of threads running on the FA_8 processor. Such a processor enables the highest thread parallelism. We estimate the ILP by measuring the average ILP in the FA_1 processor. Such a processor enables the highest ILP. Based on these measurements for a low- and a high-end machine environment, we map the applications on the chart of Figure 6-(a) and 6-(b) respectively.

We focus first on the low-end machine. The performance of

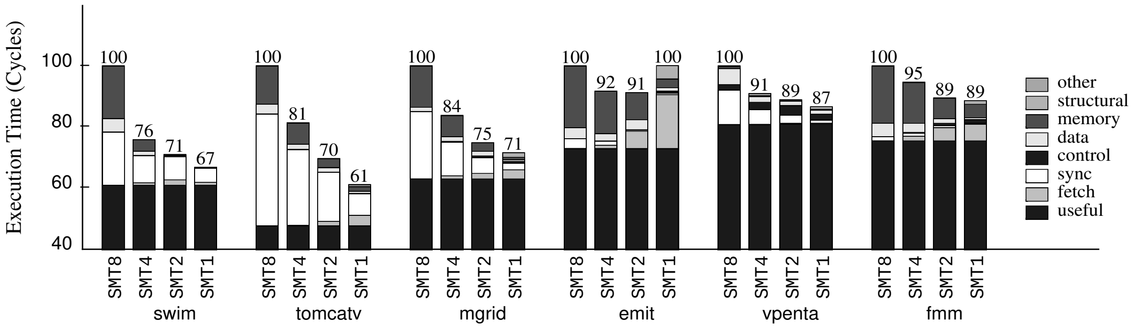


Figure 7: Comparing the centralized and clustered SMT processors for a low-end system. The chart assumes the same cycle-time for all the architectures.

the FA processors in Figure 4 is consistent with the chart in Figure 6-(a). In Figure 6-(a), we see that *ocean* and *vpenta* fall closer to the lower right corner. In Figure 4, these are the two applications where FA_8 performs best while FA_1 performs abysmally. The rest of the applications fall more to the center, so FA_4 tends to perform best in Figure 4. The leftmost point in Figure 6-(a) is *tomcatv*. This is consistent with the fact that FA_2 performs best for *tomcatv* in Figure 4. We now consider the SMT_2 processor. All the points in Figure 6-(a) fall in the center of its optimal region. Therefore, SMT_2 performs very well in Figure 4, better than the FA processors.

If we now consider the high-end system (Figure 6-(b)), we see that, compared to the low-end system, the data points have moved to the left and to the bottom in the chart. The number of threads decreases because serial sections are more important and the ILP decreases because parallel threads suffer more hazards. Focusing on the data points in the chart we see that, since *ocean* and *vpenta* have moved little, FA_8 is still the best FA processor for these two applications in Figure 5. *Swim*, *fmm* and *mgrid* have moved to the left side and, therefore, FA_1 or FA_2 work best. Finally, *tomcatv* has moved much to the left and, therefore, the best design is FA_1 in Figure 5. As for SMT_2, all the data points except for *tomcatv* still fall within its optimal region. Therefore, it continues to perform better than the best FA processor for these applications in Figure 5.

5.2 Clustered and Centralized SMT Architectures

We now justify the choice of SMT_2 in our previous section. Towards this, we evaluate the three clustered SMT and the centralized SMT_1 architectures for the low- and high-end machines. Figure 7 shows the performance of the four architectures for a low-end machine. For each application, the execution time is normalized to that of the SMT_8 processor. The SMT_8 processor is a special case of the clustered SMT processor in that it is the same as the FA_8 processor. Note that, as usual, our charts show the execution time in terms of number of cycles, without considering the different cycle-times of the different architectures. We will consider the different cycle-times later.

The figure shows that the performance of the SMT processors is usually fairly predictable across applications. It increases from SMT_8 to SMT_1. This effect, of course, is due to the higher adaptability of the wide-issue SMTs to variations in the ILP and thread parallelism of the application. For most applications, we can see that there is also a steady increase in the *fetch* hazards, going from SMT_4 to SMT_1. This is due to the unified instruction queue that we use for dynamic instruction issue and the nature of the fetch mechanism. For example, instructions issued by a thread may remain in the queue for a long time waiting on a cache miss. Since they clog the queue, other threads are unable to fetch instructions. This fetch bottleneck has been discussed in great detail by Tullsen *et al* [16]. They suggest several alternatives, such as partitioning the fetch unit or using instruction count feedback techniques to use the fetch unit more intelligently. The centralized SMT is more susceptible to this problem than the clustered SMTs.

Overall, the figure shows that the SMT_2 processor achieves nearly the same performance as the centralized SMT. The difference between the two ranges from $0 - 9\%$ normalized to SMT_8. This shows that the full issue bandwidth that is available in the centralized SMT processor is hardly used by the multithreaded applications that we used. As a result, limiting the level of ILP that may be exploited decreases the performance only to a limited extent.

Finally, figure 8 shows the performance of the four SMT processors for a high-end system. In each application, the bars are again normalized to SMT_8. The data in this figure is very similar to that in Figure 7. Therefore, the conclusions for the low-end system apply fully here. The SMT_2 architecture is again only slightly slower than SMT_1.

If we now consider the differences in clock cycle-time, the picture changes dramatically in favor of the SMT_2 architecture. According to [12], 4-issue clusters are likely to have a factor of 2 in clock frequency advantage over the centralized 8-issue cluster. This gives a very large clock frequency advantage to SMT_2 over SMT_1. Given that the SMT_2 and SMT_1 bars in Figures 7 and 8 are so close, SMT_2 is a much better design point. Furthermore, the 4-issue and the lower-issue clusters (represented by SMT_4 and SMT_8) would have similar clock frequencies with current technology [2]. Consequently, SMT_2 is the most cost-effective organization.

6 Conclusions

Since aggressive superscalar processors are becoming so complex, there is much interest in alternate designs that exploit thread-level parallelism. With future applications likely to be multithreaded, this is a very promising approach. One of the current proposals is to configure the architecture such that the chip resources are partitioned in a fixed manner between the threads. These fixed-assignment architectures (FA) risk wasting resources when one of the threads stalls due to hazards or when there is a lack of threads in the application. Although basic simultaneous multithreading (SMT) allows resources to be shared, the structure is so centralized that it is likely to slow down the clock frequency. However, given that the amount of thread- and instruction-level parallelism of applications varies widely, the flexibility of the SMT architecture is very valuable.

The architecture that we have explored in this paper is a

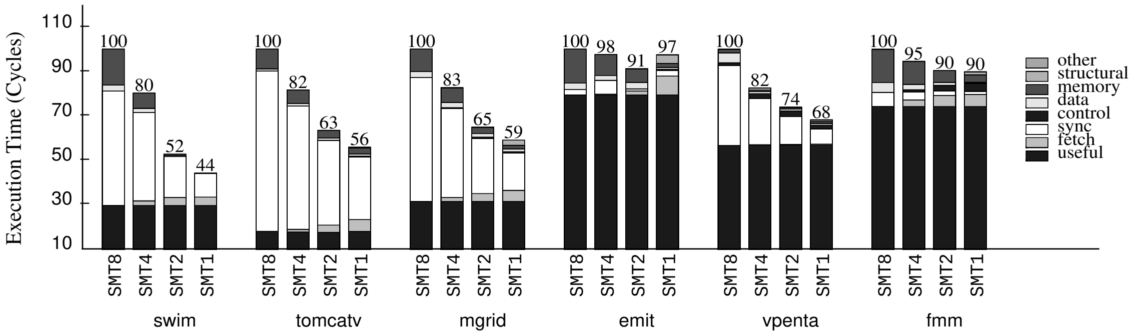


Figure 8: Comparing the centralized and clustered SMT processors for a high-end system. The chart assumes the same cycle-time for all the architectures.

hybrid of the FA and the SMT approach, namely the clustered SMT architecture. We have shown that this restricted level of simultaneous multithreading is able to capture most of the performance benefits of the fully centralized approach while, at the same time, allowing the SMT design to cycle at high frequency. Therefore, we conclude that the hybrid organization is the most cost-effective one.

Acknowledgments

We thank the referees and the members of the I-ACOMA group for their valuable feedback. Josep Torrellas is supported in part by an NSF Young Investigator Award.

References

[1] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Parallel Programming with Polaris. *IEEE Computer*, 29(12):78–82, December 1996.

[2] The 21264: A Superscalar Alpha Processor with Out-of-Order Execution. Microprocessor Forum, October 1996.

[3] P. Dubey, K. O'Brien, K. O'Brien, and C. Barton. Single-Program Speculative Multithreading (SPSM) Architecture: Compiler-assisted Fine-grained Multithreading. In *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT '95*, pages 109–121, June 1995.

[4] M. Franklin and G. Sohi. ARB: A Hardware Mechanism for Dynamic Memory Disambiguation. *IEEE Transactions on Computers*, 45(5):552–571, May 1996.

[5] M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S.-W. Liao, E. Bugnion, and M. Lam. Maximizing Multiprocessor Performance with the SUIF Compiler. *IEEE Computer*, 29(12):84–89, December 1996.

[6] V. Krishnan and J. Torrellas. Efficient Use of Processing Transistors for Larger On-Chip Storage: Multithreading. In *Workshop on Mixing Logic and DRAM: Chips that Compute and Remember*, June 1997.

[7] V. Krishnan and J. Torrellas. Executing Sequential Binaries on a Clustered Multithreaded Architecture with Speculation Support. In *4th High Performance Computer Architecture (HPCA) Workshop on Multi-Threaded Execution, Architecture and Compilation (MTEAC'98)*, February 1998.

[8] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-based Cache Coherence Protocol for the DASH Multiprocessor. In *17th International Symposium on Computer Architecture*, pages 148–159, May 1990.

[9] J. Lo, S. Eggers, J. Emer, H. Levy, R. Stamm, and D. Tullsen. Converting Thread-Level Parallelism Into Instruction-Level Parallelism via Simultaneous Multithreading. *ACM Transactions on Computer Systems*, pages 322–354, August 1997.

[10] E. Lusk et al. *Portable Programs for Parallel Processors.* Holt, Rinehart and Winston Inc., New York, 1987.

[11] K. Olukotun, B. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The Case for a Single-Chip Multiprocessor. In *7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–11, October 1996.

[12] S. Palacharla, N. Jouppi, and J. Smith. Complexity-Effective Superscalar Processors. In *24th International Symposium on Computer Architecture*, pages 206–218, June 1997.

[13] E. Rotenberg, S. Bennett, and J. Smith. Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching. In *29th International Symposium on Microarchitecture*, December 1996.

[14] G. Sohi, S. Breach, and T. Vijayakumar. Multiscalar Processors. In *22nd International Symposium on Computer Architecture*, pages 414–425, June 1995.

[15] J. Tsai and P. Yew. The Superthreaded Architecture: Thread Pipelining with Run-Time Data Dependence Checking and Control Speculation. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques (PACT '96)*, pages 35–46, October 1996.

[16] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm. Exploiting Choice: Instruction Fetch and Issue on an implementable Simultaneous Multithreading Processor. In *23rd International Symposium on Computer Architecture*, pages 191–202, May 1996.

[17] D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading: Maximizing on-chip Parallelism. In *22nd International Symposium on Computer Architecture*, pages 392–403, June 1995.

[18] J. Veenstra and R. Fowler. MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In *Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'94)*, pages 201–207, January 1994.

[19] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *22nd International Symposium on Computer Architecture*, pages 24–36, June 1995.