Hybrid Multithreading for VLIW Processors*

Manoj Gupta
mgupta@ac.upc.edu

Fermín Sánchez
fermin@ac.upc.edu

Josep Llosa
josepll@ac.upc.edu

Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
Barcelona, Spain

ABSTRACT

Several multithreading techniques have been proposed to reduce resource underutilization in Very Long Instruction Word (VLIW) processors. Simultaneous MultiThreading (SMT) is a popular technique that improves processor performance by issuing multiple instructions from different threads. In VLIW processors, SMT requires extra hardware to merge instructions from different threads. The complexity of this hardware increases substantially with the number of threads. On the other hand, techniques like Interleaved MultiThreading (IMT) do not need any merging hardware, and support a larger number of threads at reasonable cost. In this paper, we propose Hybrid MultiThreading (HMT), a technique that at each cycle merges instructions from only a subset of threads. HMT supports a reasonable number of threads with a low merging hardware cost. For instance, it is possible to support 8 hardware threads with a merging hardware for only 2 threads. The experimental results show that using HMT improves the multithreading performance significantly. Further, supporting 8 hardware threads with HMT but using a 4-thread merging hardware achieves a performance similar to merging 8 threads simultaneously with a significantly lower merging hardware cost.

Categories and Subject Descriptors

C.1.1 [**Processor Architectures**]: RISC/CISC, VLIW architectures; C.3 [**Special-Purpose and Application-Based Systems**]: Realtime and embedded systems

General Terms

Design, Performance

*This work is supported by Spanish Ministry of Science and Technology under contract CICYT TIN2007-60625, FI grant from AGAUR/Generalitat de Catalunya, SARC (Scalable computer ARChitecture) Project and HiPEAC European Network of Excellence.

.

Keywords

Clustered VLIW Processors, Multithreading

1. INTRODUCTION

Very Long Instruction Word (VLIW) processors have gained wide acceptance in the embedded domain due to hardware simplicity, low cost and low power consumption [11, 18]. To exploit high Instruction Level Parallelism (ILP), VLIWs need to be designed with a significant issue width. However, the centralized Register File (RF), with all the Functional Units (FUs) connected to it, becomes a bottleneck due to an increase in RF delay, power and area [17]. Clustered VLIW architectures have multiple RFs and cluster the Functional Units (FUs) to the RFs they are connected to. Many VLIWs have been designed using the clustered approach [11, 18].

Some applications scale well with issue width, which makes a high issue width processor desirable. However, the ILP exposed in many applications, or in some code regions, is limited and the processor is heavily underutilized. Besides, in a production environment, high ILP applications (like image processing) coexist with low ILP applications (like control code or the OS itself). In the context of VLIW architectures, processor underutilization can be characterized in terms of vertical and horizontal waste. Vertical waste are the cycles where no operations are issued at all. Horizontal waste is the underutilization of the issue width of the processor, i.e. the number of operations issued in a cycle is less than the maximum number of operations that can be issued per cycle. Both vertical and horizontal waste arise because control and data dependencies in the program limit the number of operations that can be issued in a given cycle. Besides, operations that have variable latency (for instance, memory operations) stall the processor if the actual latency is greater than the expected one, resulting in additional vertical waste.

Several multithreading techniques have been proposed to reduce the vertical and horizontal waste in the processor. Block MultiThreading (BMT) [23] executes instructions from a single thread until it is blocked by a long latency event (a cache miss, for instance). Interleaved MultiThreading (IMT) [2, 19] does a zero cycle context switch every cycle, so that instructions from different threads are interleaved at execution time. Simultaneous MultiThreading (SMT) [20] issues each cycle multiple instructions from multiple threads. In a SMT processor, issue-slots of the processor are filled by operations of different threads, converting thread level parallelism (TLP) into ILP.

Implementing SMT on VLIWs require complex structures which limits the number of threads that can be supported.

One reason for the limited scalability is the complexity of the merging hardware required to merge instructions from different threads. Cluster-level simultaneous MultiThreading (CSMT) [8] reduces the complexity of merging hardware by merging instructions at cluster-level. Compared to both SMT and CSMT, approaches like IMT scale better with number of threads in terms of hardware complexity, and can support a larger number of threads, since no merging hardware is required.

In this paper, we propose Hybrid MultiThreading (HMT). HMT combines the advantages of both IMT and SMT. HMT merges instructions from a subset of threads. Each cycle, a new subset is selected in a manner analogous to IMT. Thus, HMT can support a reasonable number of threads with a given merging hardware cost. For instance, it is possible to support 8 threads with a hardware that can merge only 2 threads at a time. This approach gets higher performance at a low merging hardware cost. HMT can use any approach to merge instructions from different threads. In particular, this paper evaluates HMT with operation-level and cluster-level simultaneous multithreading.

The rest of the paper is organized as follows. First, Section 2 describes the relevant multithreading schemes. Section 3 discusses the motivation for HMT. HMT is discussed in Section 4. The experimental setup is described in Section 5. A detailed performance evaluation is presented in Section 6. Finally, Section 7 concludes the paper.

2. RELATED WORK

2.1 Multithreading Schemes

In this section, we briefly describe the most relevant multithreading schemes proposed in the literature.

Block MultiThreading (BMT) [23] executes instructions from a single thread until it is blocked by a long latency event (a cache miss, for instance). When that happens, a fast context switch gives control to a different thread so that most of the miss latency is hidden.

Interleaved MultiThreading (IMT) [2, 19] does a zero cycle context switch every cycle, so that instructions from different threads are interleaved at execution time. IMT allows the removal of the bypass network and interlocking hardware, since control and data dependencies between the instructions in the pipeline are eliminated when the number of interleaved threads is greater than or equal to the number of pipeline stages. However, doing so hinders single thread performance when only one thread is present. A simple modification, where any thread that produces a cache miss is marked as blocked and instructions from only non-blocked threads are issued, combines the best of both IMT and BMT. This variation of IMT is used in this paper for evaluations.

Simultaneous MultiThreading (SMT) [20] issues instructions from multiple threads each cycle in contrast to BMT and IMT where, at a given time, instructions from only one thread are issued. In a SMT processor, issue-slots of the processor are filled by operations from different threads, converting thread level parallelism (TLP) into ILP. As a SMT processor simultaneously reduces both horizontal and vertical waste, the resource usage is much more efficient than in IMT and BMT.

An example of the different multithreading approaches viz. BMT, IMT and SMT is shown in figures 1(a), 1(b) and 1(c) respectively for two threads A and B on a 4-issue VLIW processor. In the figures, each box in a row represents an issue slot. A box with a label indicates that the slot is in use by that particular thread and an empty box represents an unused issue slot. BMT and IMT can reduce vertical waste by issuing instructions from different threads, but not horizontal waste. In BMT, a few vertical slots are still wasted due to context switch time. Since IMT also reduces short stalls, IMT performance, in general, is a good upper bound for BMT as well. In contrast to BMT and IMT, SMT also removes horizontal waste in the processor. To do so, it requires an extra merging hardware that can merge instructions from different threads.

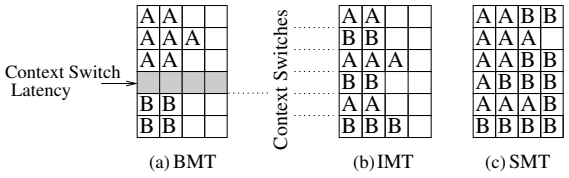


Figure 1: Comparison of Multithreading Schemes

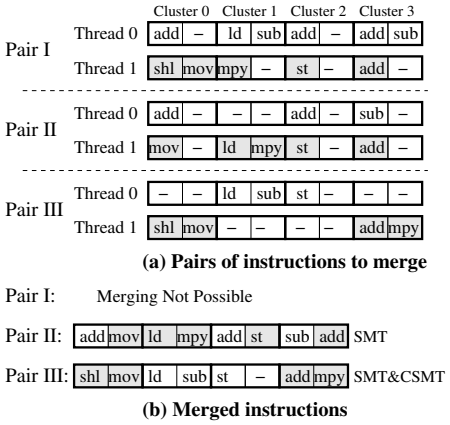


Figure 2: Instruction Merging in SMT and CSMT

Cluster-level Simultaneous MultiThreading (CSMT) [8] is a variant of SMT specifically proposed for clustered VLIW processors. In CSMT, the instruction merging is done at a cluster-level granularity instead of the operation-level merging done by SMT. Hence, CSMT issues instructions from multiple threads only when the threads use different clusters. This restricts the opportunities to merge the instructions in comparison to SMT, but has a lower complexity implementation of the merging hardware.

To illustrate how instructions from different threads are merged in CSMT, Figure 2(a) displays 3 pairs of instructions for 2 threads for a 4-cluster 2-issue per cluster (8-issue) architecture. In the figure, operations in the white background belong to Thread 0 and operations with a grey background belong to Thread 1. Note that when two instructions are merged, they have to be merged in their entirety i.e. it is not possible to choose only a non-conflicting part of an instruction because doing so breaks VLIW execution semantics. Also, if a pair of instructions can be merged by CSMT, it can always be merged by SMT but not vice-versa. Also note that if both CSMT and SMT can merge a pair of instructions, the final merged instruction is identical for both SMT

and CSMT. The final instructions obtained by merging are shown in Figure 2(b). Neither CSMT nor SMT can merge Pair I because of conflicts at clusters 0, 1 and 3, both at operation-level and cluster-level. Pair II can be merged by SMT since there are no conflicts at operation-level. CSMT, however, cannot merge this pair, since both instructions in the pair use clusters 0, 2 and 3. As CSMT checks resource conflicts at the cluster-level, there is a conflict at these clusters. Pair III, however, can be merged by CSMT (and SMT as well) as the first instruction uses only clusters 1 and 2 which are not used by the other instruction.

Next we discuss several multithreading schemes and enhancements which are relevant to our proposals.

2.2 Other Related Work

To improve multithreading performance in clustered VLIWs, a technique named Cluster Renaming is proposed in [8]. The authors observe that there is a resource usage bias in most of the programs with cluster 0 being the most heavily used. This bias reduces the opportunities for merging instructions from different threads. Cluster renaming solves this problem by statically distributing the clusters of each thread which reduces the contention for the clusters. The renaming mechanism is simply a rotation of the the original cluster assignment done by compiler by a given renaming value. The renaming value of each thread is a fixed number computed at design time, based on the number of clusters and the number of simultaneous threads supported by the processor. For instance, in a 2-thread 4-cluster machine, Thread 0 is rotated by 0 and Thread 1 is rotated by 2. On a 8-thread 4-cluster machine, Thread 0 is rotated by 0, Thread 1 by 1, Thread 2 by 2, Thread 3 by 3, Thread 4 by 0, and so on. Note that with 8 threads, some threads share the same renaming value since there are more threads than the number of clusters. Cluster renaming is used in all our experiments.

M-Machine [5] uses IMT to run several applications at the same time. It also uses compiler generated threads to improve single application performance. In M-Machine, either complete VLIW instructions are issued from a single thread or short instructions are issued from multiple compiler generated threads by executing each thread in a different cluster. In the latter case, threads have been compiled to use only a single cluster. A similar approach, Multithreaded extension to clustered VLIWs [3] also uses compiler generated threads to run multiple threads on a clustered VLIW processor.

Balanced MultiThreading [21] is a related multithreading technique proposed for high end out-of-order superscalar processors. Balanced multithreading combines the SMT ability with Block MultiThreading. To do so, extra thread contexts are stored in a special storage location on-chip. If a running thread encounters a L2 cache miss, it is swapped with one of the extra contexts. Use of balanced multithreading saves the register file space requirement for the extra thread contexts but several cycles are required for context swapping.

Subset Static Interleaved Multithreading (SSIMT) [12] is a technique similar to Balanced MultiThreading. SSIMT combines Block MultiThreading with IMT for embedded VLIW Processors. SSIMT maintains several background thread contexts on-chip. The background threads are swapped with a running thread if that thread encounters a cache miss at a low swapping penalty. Both Subset Static Interleaved Multithreading and Balanced multithreading proposals are orthogonal to Hybrid MultiThreading, the approach presented in this paper.

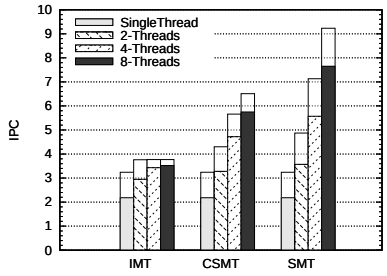


Figure 3: Average IPC for IMT, CSMT and SMT

3. MOTIVATION

Supporting a larger number of threads is a possible way for improving performance. However, some multithreaded schemes do not benefit from an increase beyond a small number of threads. To illustrate this, Figure 3 shows the average IPC obtained by IMT, CSMT and SMT for the workloads evaluated in this paper (explained later in Section 5) on a single-thread, 2-thread, 4-thread and an 8-thread machine. In the figure, the filled portion of the bars is the IPC obtained for the real memory system (described in Section 5) and the extra white bar on top represents the additional IPC obtained for a perfect memory model assuming no cache misses. For an ease of comparison, the single-thread IPC bar is shown for all IMT, CSMT and SMT.

The first thing to notice is that even with a perfect memory model, IMT does slightly better than the single-thread processor. This is because, despite the fact that there is no vertical waste due to memory stalls, a few issue cycles are still lost due to taken branches and def-to-use latency of operations like loads, multiplies and comparisons. IMT also hides this vertical waste by issuing instructions from an alternate thread. However, little improvement in performance is achieved in moving from a 2-thread machine to a 4-thread or an 8-thread machine. With a real memory system, a moderate performance difference still exists between a 2-thread and a 4-thread machine, but an 8-thread machine provides only marginal performance improvement over a 4-thread machine. This happens because IMT can only remove vertical waste. Vertical waste keeps on reducing with an increase in number of threads. Thus, IMT gets little benefit by supporting more than 4 threads. In contrast to IMT, CSMT and SMT also remove horizontal waste, and thus, both CSMT and SMT performance keep on improving significantly up to 8 threads. In moving from a 4-thread machine to an 8-thread machine, CSMT performance improves by 15% while SMT performance improves by 29% for the perfect memory model, while for the real memory system, there is a performance improvement of 22% for CSMT and 37% for SMT.

Supporting a large number of threads on a SMT or CSMT processor is desirable because of the performance gains that are achieved. However, the hardware complexity increases with the number of threads and limits the number of threads that can be supported simultaneously. In general, most of the hardware complexity can be attributed to the extra reg-

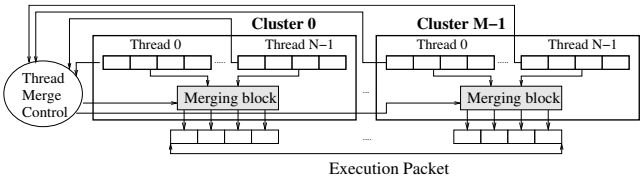
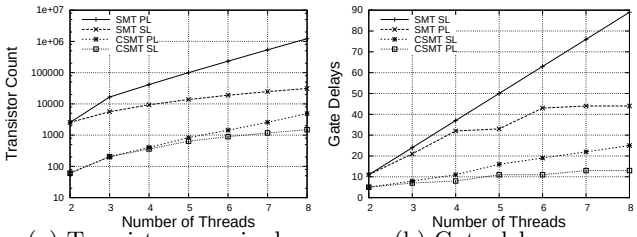


Figure 4: Merging Hardware for a 4-issue per cluster Processor



(a) Transistors required (b) Gate delays

Figure 5: CSMT and SMT thread merge control cost

ister sets required to support multiple threads and the merging hardware required to merge instructions from different threads. Most of the multithreading schemes including IMT, SMT, CSMT, etc. require a register set per thread. The cost of extra register sets is not trivial. Nevertheless, systems like Cray MTA/Tera [2] had 128 register sets per processor to support 128 threads using IMT with a VLIW ISA. Compared to extra register sets, which has the same cost for all multithreading schemes, the cost of merging hardware varies significantly depending on the multithreading scheme used.

3.1 Merging Hardware

This section compares the merging hardware for the multithreading schemes IMT, CSMT and SMT. Figure 4 shows the merging hardware required for a N-Thread M-cluster 4-issue per cluster processor. The merging hardware consists of 2 parts, a thread merge control and a merging block per cluster. Thread merge control takes the resource usage information of the instructions as input and generates control signals for the merging block. The merging uses these control signals to generate the execution packet by merging the instructions of the threads. Next we discuss the implementation of the merge control and merging block for IMT, CSMT and SMT.

IMT thread merge control selects a different thread at each cycle. Hence IMT thread merge control can be simply implemented as a counter which is incremented at each cycle. The merging block for IMT is simply a multiplexer at each cluster. The output of the thread merge control is used by all the multiplexers to select the instruction of a thread to be issued as the execution packet.

In comparison to IMT, CSMT can issue instructions from multiple threads at a time. Hence, CSMT also requires to check for resource collisions at cluster-level amongst the threads. This results in a more complex thread merge control than for IMT. However, the cost of the merging block for CSMT is the same as for IMT, since CSMT also requires only one multiplexer per cluster (each multiplexer is driven by different control signals).

SMT checks resource collisions[1] at operation-level and can select operations from multiple threads at the same cluster, in contrast to CSMT. To fit operations from multiple threads in the same cluster, SMT merging hardware needs to route the operations of the instructions. SMT thread merge control also generates the appropriate signals for routing the operations. The merging block uses these routing signals to produce the final execution packet. The thread merge control is more complex for SMT than CSMT. However, the area required by the merging block to do the routing is similar to the area required for the multiplexers in both IMT and CSMT, assuming the methodology used in [14] for interconnect area computation. This is because the number of input and output wires for the merging block is the same as that of the multiplexers.

For all the 3 schemes considered namely IMT, CSMT and SMT, the cost of the thread merge control is the only variable cost in the merging hardware. Thus, the cost of the thread merge control is the only factor that influences scalability of CSMT and SMT compared to IMT. Next, we discuss this cost.

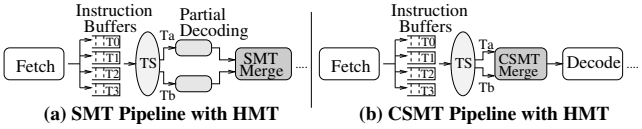
3.2 Thread Merge Control Cost Analysis

Two implementations of thread merge control for CSMT and SMT [9] are considered in this paper viz. serial and parallel. The serial implementation is a cascading logic checking a different thread at each level. The parallel implementation, on the other hand, checks all the possible thread selections in parallel and selects the one conforming to the selection policy. The parallel implementation has lower delay than the serial one but has a much higher hardware overhead, which grows exponentially with number of threads.

Figure 5 shows the cost of the thread merge control with varying number of threads for a 4-cluster 4-issue per cluster architecture for both CSMT and SMT. Figure 5(a) shows the cost in terms of number of transistors required on a logscale, and Figure 5(b) shows the cost in terms of gate delays. In the figures, labels 'CSMT PL' and 'CSMT SL' refer to the parallel and serial implementations of CSMT thread merge control, while labels 'SMT PL' and 'SMT SL' refer to the parallel and serial implementations of SMT thread merge control. The values for CSMT have been taken from [9]. The computation details of the gate delays and transistor count for SMT thread merge control are computed following the same methodology as in [9] and are omitted from this paper for space reasons. As shown in the figures, the complexity of the thread merge control for SMT increases significantly with the number of threads, both in terms of transistors required and gate delays, and constrain its scalability. In particular, the increase in gate delays limits the scalability of SMT to 2 threads. Higher delays in thread merge control can be tolerated by implementing it using extra pipeline stages. Increasing pipeline stages, however, degrades single-thread performance significantly when only one thread is present, and may not be acceptable.

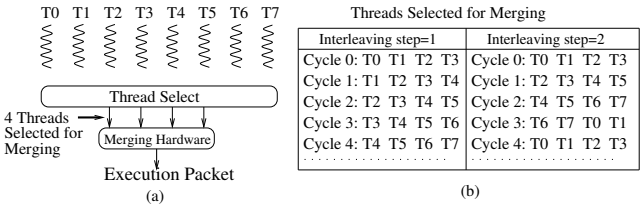
Compared to SMT, CSMT scales better with number of threads. However, despite having lower complexity than

[1]In most VLIW processors, certain type of operations can be executed only at fixed issue slots. For instance, in our base architecture, while ALU operations may be executed at any issue slot, operations like memory load/store, multiply and branch can only be executed at their fixed slots.



(a) SMT Pipeline with HMT (b) CSMT Pipeline with HMT

Figure 6: HMT Pipeline



Interleaving step=1	Interleaving step=2
Cycle 0: T0 T1 T2 T3	Cycle 0: T0 T1 T2 T3
Cycle 1: T1 T2 T3 T4	Cycle 1: T2 T3 T4 T5
Cycle 2: T2 T3 T4 T5	Cycle 2: T4 T5 T6 T7
Cycle 3: T3 T4 T5 T6	Cycle 3: T6 T7 T0 T1
Cycle 4: T4 T5 T6 T7	Cycle 4: T0 T1 T2 T3

Figure 7: Hybrid Multithreading Example

SMT, CSMT still suffers from scalability issues. While gate delays is an issue for the serial design, the parallel design requires a large number of transistors for large number of threads. Thus, none of these approaches is suitable for supporting a large number of threads.

4. OUR PROPOSAL: HYBRID MULTITHREADING

Hybrid MultiThreading (HMT) combines the advantages of interleaving threads and executing simultaneously instructions from different threads. HMT merges, simultaneously, instructions from only a subset of threads instead of merging all the threads. Every cycle, a new subset is selected in a manner analogous to IMT. If one or more threads produce a cache miss (or even a short latency delay that arises because of data dependencies), selecting a subset of threads helps in hiding the latency efficiently. Moreover, the merging hardware can still operate efficiently even if a few threads are blocked. The number of threads that can be merged at a time is dictated by the merging hardware cost. However, HMT approach supports a larger number of threads without incurring a large merging hardware cost. For instance, 4 threads can be supported with a merging hardware that can merge only 2 threads at a time. HMT is independent of the implementation of the merging hardware. Either SMT or CSMT can be used to merge instructions from different threads. Figures 6(a) and (b) show the first pipeline stages for both SMT and CSMT processors with HMT respectively, where 4 threads are supported but only 2 are merged at a time by selecting a subset of 2 threads each cycle. In the figure, TS represents the Thread Select which selects a subset of threads for further execution. For a SMT processor, using a subset of threads also means that fewer threads need to be partially decoded (SMT requires partially decoded operation to recognize the FU used).

Figure 7 shows an example of HMT. The HMT processor shown in the figure 7(a) supports eight threads, T0-T7, but the merging hardware has the ability to merge instructions from only 4 threads at a time. Thus, up to 4 threads are selected every cycle by the Thread Select. The merging hardware takes the selected threads and produces an execution packet by merging their instructions. Note that the selection of a thread does not guarantee its inclusion in the execution packet. The selection only implies that the thread's instruction will be considered for merging. The actual inclusion in the execution packet depends on the merging scheme used by the merging hardware and whether the thread is blocked or not. Every cycle, a new subset of threads is selected which is decided by the interleaving step. Interleaving step is the number of threads skipped to select the new set of threads. Following section discusses interleaving step in detail.

4.1 Interleaving Step

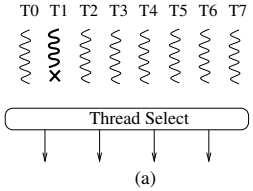
The first step in HMT is to select the subset of the threads that should be considered for merging in next cycle. The primary factor that influences the selection of threads is the value of interleaving step used. IMT, for instance, uses always an interleaving step of 1 and selects the next thread every cycle. In HMT, using different values for interleaving step creates interesting thread selections which may obtain different performance. For example, for the 8-thread HMT machine with a 4-thread merging hardware shown in figure 7(a), with an interleaving step of 1, the 4 threads selected at a given cycle contain 3 threads that were also selected in the previous cycle. With an interleaving step of 2, the selected threads contain only 2 threads that were also selected in previous cycle. Figure 7(b) shows the thread selections obtained by using different values of interleaving step of 1 and 2. Initially, threads T0-T3 are selected at cycle 0 for all examples. With an interleaving step of 1 (column 1 of Figure 7(b)), the first thread T0 is skipped at cycle 1, and the next 4 threads T1-T4 are selected, and so on. With an interleaving step of 2, first two threads T0 and T1 are skipped at cycle 1, and the next 4 threads T2-T5 are selected. At cycle 2, threads T2 and T3 are skipped and threads T4-T7 are selected, and so on.

Thread Select (TS) initially considers the selections obtained by the interleaving. However, some of the threads in a given selection may be blocked, lowering the opportunities for merging. Following section discusses several thread selection schemes that can be employed by the TS to replace a blocked thread.

4.2 Thread Selection Schemes

Thread selection can be divided into two categories: Static and Dynamic. In a static thread selection, TS does not take any runtime decisions and completely relies on interleaving for selecting the threads. In Dynamic thread selection, the subset of threads obtained using interleaving is initially considered, but if some of the threads in the subset are blocked (because of a cache miss, for instance), they can be replaced by other threads. Dynamic thread selection is more complex than a static one, but allows a better use of the resources. In particular, an IMT implementation where the blocked threads are skipped also belongs to dynamic thread selection category, as instructions from a non-blocked thread are issued if the given thread is blocked.

Figure 8 shows a comparison of a static vs dynamic thread selection for a machine with 8 threads (T0-T7) and a thread select that can select up to 4 threads. An interleaving step of 1 is assumed in this example. As shown in the Figure 8(a), Thread T1 is assumed to be blocked. Static thread selection selects Thread T1 at both cycle 0 and cycle 1 despite T1 being blocked as shown in Figure 8(b). Thus, a selection slot is wasted at these cycles. Dynamic thread selection, however, skips Thread T1 and instead selects the non-blocked

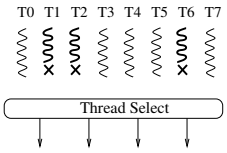


(a)

	Static Selection	Dynamic Selection
Cycle 0	T0 ~~T1~~ T2 T3	T0 T2 T3 T4
Cycle 1	~~T1~~ T2 T3 T4	T2 T3 T4 T5
Cycle 2	T2 T3 T4 T5	T2 T3 T4 T5
Cycle 3	T3 T4 T5 T6	T3 T4 T5 T6
.....	.....	.....

(b)

Figure 8: Static vs Dynamic Thread Selection

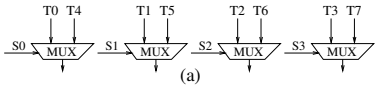


(a)

	Selected Threads			
	Static	FNB	Eq	Eq+FNB
Cycle 0	T0 ~~T1~~ ~~T2~~ T3	T0 T3 T4 T5	T0 T5 ~~T2~~ T3	T0 T5 T4 T3
Cycle 1	~~T1~~ ~~T2~~ T3 T4	T3 T4 T5 T7	T5 ~~T2~~ T3 T4	T5 T3 T4 T7
Cycle 2	~~T2~~ T3 T4 T5	T3 T4 T5 T7	~~T2~~ T3 T4 T5	T3 T4 T5 T7
Cycle 3	T3 T4 T5 ~~T6~~	T3 T4 T5 T7	T3 T4 T5 ~~T6~~	T3 T4 T5 T7
Cycle 4	T4 T5 ~~T6~~ T7	T4 T5 T7 T0	T4 T5 ~~T6~~ T7	T4 T5 T7 T0

(b)

Figure 9: Thread Selection Schemes



(a)

	Interleaving step=1					Interleaving step=2					Interleaving step=4				
	Threads				S0 S1 S2 S3	Threads				S0 S1 S2 S3	Threads				S0 S1 S2 S3
Cycle 0	T0	T1	T2	T3	0 0 0 0	T0	T1	T2	T3	0 0 0 0	T0	T1	T2	T3	0 0 0 0
Cycle 1	T4	T1	T2	T3	1 0 0 0	T4	T5	T2	T3	1 1 0 0	T4	T5	T6	T7	1 1 1 1
Cycle 2	T4	T5	T2	T3	1 1 0 0	T4	T5	T6	T7	1 1 1 1	T0	T1	T2	T3	0 0 0 0
Cycle 3	T4	T5	T6	T3	1 1 1 0	T0	T1	T6	T7	0 0 1 1	T4	T5	T6	T7	1 1 1 1
Cycle 4	T4	T5	T6	T7	1 1 1 1	T0	T1	T2	T3	0 0 0 0	T0	T1	T2	T3	0 0 0 0

(b)

Figure 10: Static Thread Selection Hardware

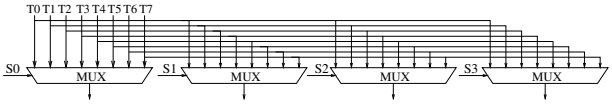


Figure 11: General Dynamic Thread Selection Hardware

threads T4 at cycle 0 and T5 at cycle 1. Thus, all selection slots are completely utilized. The following dynamic thread selection schemes have been considered in this paper:

First Non Block (FNB): If one of the threads in the set considered for merging is blocked, then it is substituted by the first non-blocked thread. This scheme is used in the example in Figure 8(b).

Equivalent (Eq): This scheme is a consequence of cluster renaming. The use of cluster renaming creates an interesting case: If the number of threads is greater than the number of clusters, multiple threads share the same renaming value. To try to maximize the number of instructions that can be merged, the thread selection scheme should avoid the selection of threads that have the same renaming value. Using FNB scheme, however, may result in cases where multiple threads in the selected subset have the same renaming value, resulting in an increased contention for resources for some clusters. For instance, using the same example as in Figure 8(b), where Thread T1 is blocked, and assuming a 4-cluster machine, FNB scheme selects at cycle 0 Thread T4 as the replacement for Thread T1. Thus, threads T0, T2, T3 and T4 are selected to be merged. However, Thread T4 and Thread T0 have the same renaming value. Thus, both threads probably will compete for resources in same clusters. This limits the benefits of selecting a replacement thread. An intelligent choice would be to pick Thread T5 as the replacement for Thread T1, as both of them have the same renaming value. Then, if T0, T2, T3 and T5 (instead of T4) are the selected threads, none of them have the same renaming value. Hence, a better merging of the instructions from these threads can be expected.

Equivalent + First Non Block (Eq+FNB): This scheme is a combination of Eq and FNB schemes. If a thread is blocked, first an equivalent thread is considered as the replacement. If the equivalent thread is also blocked, then the first thread which is not blocked is used as a replacement for the blocked thread.

To illustrate different thread selection schemes, Figure 9(b) shows the thread selections done by Static, FNB, Eq and Eq+FNB schemes for an 8-thread 4-cluster machine with a 4-thread merging hardware. For all the schemes, an interleaving step of 1 is used. Threads T1, T2 and T6 are assumed to be blocked. At cycle 0, all the proposed schemes start with threads T0, T1, T2 and T3 as the initial subset. In static selection, since threads T1 and T2 are blocked, two selection slots are wasted. FNB skips the blocked threads and thus 4 non-blocked threads T0, T3, T4 and T5 are selected. Eq scheme selects Thread T5 instead of Thread T1, which is blocked, but cannot replace Thread T2 since its equivalent Thread T6 is also blocked. This results in an unutilized selection slot. Eq+FNB selects Thread T5 in place of Thread T1. Since both Thread T2 and its equivalent Thread T6 are blocked, Eq+FNB selects the first non-blocked Thread T4 as the replacement. At cycle 1, using an interleaving step of 1, the thread selection for all the schemes starts from threads T1, T2, T3 and T4 and so on.

We now discuss the relative strengths and weaknesses of both static and dynamic thread selection schemes. Figure 10(a) shows a very simple implementation for the static thread selection scheme for an 8-thread architecture where the merging hardware can support 4 threads. In the figure, T0-T7 represent the 8 threads and S0-S3 are the select signals for the muxes. This design can be used with different values of interleaving step by using a different select logic for muxes. Figure 10(b) shows the corresponding select signals for the muxes with different values of interleaving step and the selected threads. The select signals can either be stored on-chip in a table or can be generated by an on-chip logic. For instance, for an interleaving step of 1, a simple 4-bit twisted ring counter can be used.

In a dynamic thread selection, any thread can be selected at any selection slot. For instance, in FNB scheme there is no restriction in the selection. Hence, for each selection slot, all threads have to be checked, as shown in Figure 11. However, each input line to a mux is a complete VLIW instruction. Hence, the muxes themselves consume significant area and power as a lot of wiring and routing needs to be done. The select signal generation for the muxes is also more complex than for static selection, as the blocked threads need to be skipped. This adds to the delay of the merging hardware, which already has a high delay. Since the thread select should fit in the same pipeline stage as merging hardware, implementing a dynamic thread selection scheme may not be practical.

However, the Eq thread selection scheme restricts the threads that can be used for replacement. In Eq scheme, if a thread

is blocked, it can be substituted only by a fixed equivalent thread. For instance, with a 8-thread 4-cluster machine and a merging hardware of 4 threads, Thread 0 can be substituted only by Thread 4, Thread 1 by Thread 5, and so on. It is possible that more than 2 threads have the same renaming value. For instance, for a 2-cluster 8-thread machine, 4 threads have the renaming value of 0, and rest 4 have the renaming value of 1. However, with Eq scheme, we restrict to one the number of threads that are considered as replacement. This results in a design with significantly lower complexity in comparison to the general dynamic thread selection scheme. In fact, the same simple design used in static thread selection shown in Figure 10(a) can be used to implement Eq scheme. The generation of the select signals for the muxes is different though, as the blocked state of the threads has to be considered. Nevertheless, the complexity and delay of Eq thread selection scheme is similar to static thread selection. Note that the scheme Eq+FNB requires a selection hardware similar to the general dynamic thread selection presented at Figure 11.

4.3 Thread Merge Policy

HMT assigns a different priority to each hardware thread in a round robin way every cycle. The execution packet is formed by merging instructions from the selected threads according to their priority. First, the instruction from the highest priority thread is selected; then, the instruction from the next highest priority thread is selected to be merged in the execution packet if it does not collide with the already formed packet, and so on.

5. EXPERIMENTAL SETUP

The HMT evaluation done in this paper is based on the VEX clustered architecture [22] modeled upon the commercial HP/ST Lx [4] VLIW family. The VEX C compiler [22] used in this study is a derivation of the HP/ST ST200 C compiler, which itself is a derivative of the Multiflow compiler [16] that uses *Trace Scheduling* [6] as global scheduling algorithm.

VEX is a 32-bit clustered integer VLIW architecture that provides scalability of issue-width and functionality. FUs within a cluster can access only local register files with the exception of Branch FU, which may read registers from other clusters. Clusters are architecturally visible and require explicit inter-cluster copy operations to move data across them. VEX is a *less-than-or-equal* machine i.e. the actual latency of any FU can be shorter than the compiler assumption. No interlocks are required if hardware can complete an operation in the same or fewer cycles. However, for operations like memory accesses, which may take longer than the assumed latency, execution is stalled until the architectural assumptions hold true. Each cluster has 2 multipliers and 1 load/store unit, and the number of ALUs is the same as the issue width of the cluster (4 in our experiments). Memory and multiply operations have a latency of 2 cycles, and the rest have single-cycle latency. There is no branch predictor and fall-through path is the predicted path. The incorrect instructions issued following a taken branch are squashed. Compare and branch is done as a pair of operations: first operation does the comparison and sets the branch registers ahead of the actual branch, and the second is the actual branch operation. There is a 2-cycle delay from compare to branch, and the taken branch penalty is 1 cycle. A trace based in-house simulator is used to simulate a multithreaded VLIW processor.

Table 1: Benchmarks

Benchmarks	*ILP Degree*	*Description*	IPC_r	IPC_p
mcf	l	Minimum Cost Flow	0.98	1.36
bzip2	l	Bzip2 Compression	0.94	1.05
blowfish	l	Encryption	1.11	1.47
gsmencode	l	GSM Encoder	1.07	1.07
g721encode	m	G721 Encoder	1.75	1.76
g721decode	m	G721 Decoder	1.75	1.76
cjpeg	m	Jpeg Encoder	1.13	1.66
djpeg	m	Jpeg Decoder	1.76	1.77
imgpipe	h	Imaging pipeline	3.81	4.05
colorspace	h	Colorspace Conversion	5.47	8.88
x264	h	H.264 Encoder	3.89	4.04
idct	h	Inverse DCT	4.79	5.27

Experiments have been done in a 16-issue, 4-cluster architecture configuration (i.e. 4-issue per cluster). All the experiments have been done for a perfect memory model with no cache misses and for a real memory model assuming a 64KB, 4-way set-associative, 20-cycles miss penalty design for both ICache and DCache (assuming a processor frequency of 400[2] MHz. and a worst case DRAM latency of 50 ns for critical word transfer).

We have used a set of MediaBench [15] and a couple of SpecInt 2000 [10] applications which we feel are relevant for the embedded domain. We have also included production color space conversion [1], imaging pipeline [22] used in high performance printers, inverse discrete cosine transform (used in various codecs) [13] and H.264 encoder [24]. The benchmarks are shown in Table 1. Columns IPC_r and IPC_p show, for each benchmark, the average IPC for real and perfect memory models respectively. Benchmarks are classified by their IPC_p in three categories: high IPC (colorspace, imgpipe, idct and x264), medium IPC (g721encode, g721decode, cjpeg and djpeg) and low IPC (mcf, bzip2, blowfish and gsmencode). This classification is shown in column *ILP Degree* as l (low IPC), m (medium IPC) and h (high IPC).

The workload configurations used for evaluation are listed in Table 2. In order to select appropriate thread configurations, we have combined benchmarks with different IPC degrees, attempting to cover representative combinations. Column labeled as *ILP Comb* indicates these IPC combinations. For example, configuration `llmmmmhh` in Table 2 has two benchmarks with low IPC, four benchmarks with medium IPC and two benchmarks with high IPC, configuration `llllmmhh` has four benchmarks with low IPC, two benchmarks with medium IPC and two benchmarks with high IPC. The two configurations `llllllll` (all low IPC benchmarks) and `hhhhhhhh` (all high IPC benchmarks) are special configurations to evaluate the extreme cases.

We carried out the experiments by arranging the workloads in a multitasking environment. The number of threads supported by the processor is exposed as virtual CPUs and the OS task scheduler schedules as many threads to run as the number of virtual CPUs, with a timeslice of 5 million cycles. After the expiry of the timeslice, a context switch takes place and the running threads are replaced by other threads from the workload. The delay of a context switch is assumed to be negligible. For a single-thread processor, the threads run in serial order with a single thread running in the whole timeslice. For a 2-thread processor, 2 threads are scheduled to run together in the same timeslice, for a 4-thread proces-

[2]Frequency of the fastest processor, ST231, in ST200 family

Table 2: Workload configurations

ILP Comb	*Thread 0*	*Thread 1*	*Thread 2*	*Thread 3*	*Thread 4*	*Thread 5*	*Thread 6*	*Thread 7*
llllllll	181.mcf	256.bzip2	blowfish	gsmencode	181.mcf	256.bzip2	blowfish	gsmencode
llllmmmm	181.mcf	256.bzip2	blowfish	gsmencode	cjpeg	g721encode	g721decode	djpeg
llllmmhh	181.mcf	256.bzip2	blowfish	gsmencode	g721decode	djpeg	colorspace	imgpipe
llllhhhh	181.mcf	256.bzip2	blowfish	gsmencode	colorspace	imgpipe	idct	x264
llmmhhhh	181.mcf	256.bzip2	g721encode	cjpeg	x264	idct	colorspace	imgpipe
llmmmmhh	blowfish	gsmencode	cjpeg	g721encode	g721decode	djpeg	idct	x264
mmmmhhhh	cjpeg	g721encode	g721decode	djpeg	idct	x264	colorspace	imgpipe
hhhhhhhh	x264	idct	colorspace	imgpipe	x264	idct	colorspace	imgpipe

sor, 4 threads share the timeslice, and for a 8-thread processor all 8 threads share the timeslice. To improve fairness and alleviate any bias, replacement threads are picked at random from the workload, after the context switch. The workloads are executed till one thread completes executing 200 million VLIW instructions (1 VLIW instruction = 1 to 16 RISC instructions). If any of the benchmarks finishes before some thread can finish executing 200M VLIW instructions, then that benchmark is respawned again. All benchmarks except mcf and bzip2 are relatively short (between 30-100M VLIW instructions) and run to completion atleast twice.

6. RESULTS

This section presents the performance results obtained by using HMT with both operation-level (SMT) and cluster-level (CSMT) merging approaches. This section also evaluates the influence of the thread selection schemes on HMT performance. HMT performance results are shown in the figures 12 to 14. In the figures, a HMT approach with CSMT merging hardware is referred as HCSMT, while a HMT approach with SMT merging hardware is referred as HSMT. An extra label of the format $A : B$ is appended to all multi-threading configurations, where A is the number of threads that can be merged by the merging hardware and B is the number of the threads supported by the processor (virtual CPUs). A and B values are always the same for a pure CSMT and SMT machine, but are different for a HMT machine since the number of threads merged is lower than the number of threads executed. Another label, indicating the thread selection policy, is appended to HMT configurations. For instance, label "CSMT 4:4" indicates a pure 4-thread CSMT machine (4 threads supported and also merged at a time), and label "HCSMT 4:8 FNB" states a HMT machine that supports 8 threads using the FNB thread selection policy with a CSMT merging hardware that can merge 4 threads at a time. In all the figures, the filled portion of the bars is the IPC obtained for the real memory system, and the extra white bar on top represents the additional IPC obtained while using the perfect memory model. In all the experiments shown in this section, an interleaving step of 1 is used. We repeated the experiments with varying degree of interleaving step. However, little difference in HMT performance was observed. Hence, we restrict the results presented to an interleaving step value of 1.

6.1 Performance Evaluation of HMT with FNB Thread Selection Policy

Figures 12(a) and (b) show the performance comparison between HMT with the FNB thread selection policy and a pure SMT/CSMT architecture. The figures also include the performance of a single-thread machine and the peak performance achievable ("CSMT 8:8" and "SMT 8:8"). In pure CSMT and SMT machines, a significant amount of performance is lost because of cache misses. Even short latencies block threads from executing, which further results into a reduction in the number of threads that can be merged during this time. HMT mitigates this performance loss because of its ability to issue instructions from different threads every cycle. As a consequence, HMT approaches significantly improve the performance over a pure CSMT/SMT machine when using the same merging hardware. HMT achieves a significant improvement in performance even with a perfect memory model because of its ability to hide short latencies.

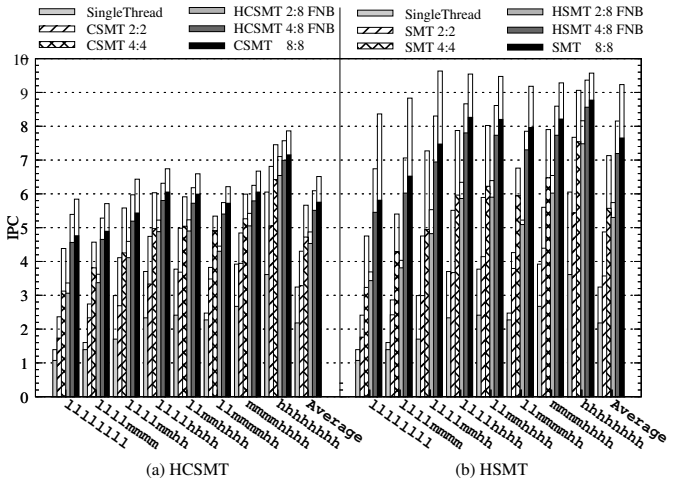


Figure 12: IPC of the workloads for HMT with FNB policy

On an average, for a CSMT merging hardware, using HMT improves the performance over pure CSMT by 38% with a 2-thread merging hardware ("HCSMT 2:8 FNB" vs "CSMT 2:2"), and 17% with a 4-thread merging hardware ("HCSMT 4:8 FNB" vs "CSMT 4:4") with the real memory model. For the SMT merging hardware, there is a performance improvement of 48% over pure SMT with a 2-thread merging hardware ("HSMT 2:8 FNB" vs "SMT 2:2"), and 29% over pure SMT with a 4-thread merging hardware ("HSMT 4:8 FNB" vs "SMT 4:4"). Further, the performance of a 4-thread merging hardware by using HMT is quite close to the peak performance of 8-thread merging hardware in pure CSMT and SMT ("CSMT 8:8" and "SMT 8:8"). On an average, with a CSMT merging hardware, "HCSMT 4:8 FNB" performance is within 4.3% of the the "CSMT 8:8" performance, while with a SMT merging hardware, "HSMT 4:8 FNB" performance is within 6.3% of "SMT 8:8" performance.

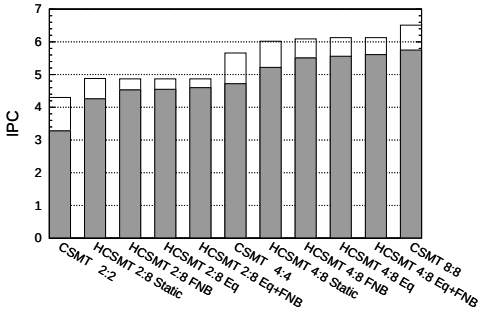


Figure 13: Average Performance of Different Thread Selection Schemes for HCSMT

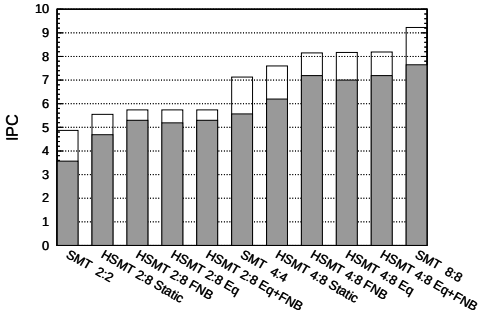


Figure 14: Average Performance of Different Thread Selection Schemes for HSMT

Besides, HMT with a 2-thread merging hardware achieves performance close to pure 4-thread CSMT/SMT configurations (within 4% for CSMT merging hardware and 4.8% for SMT merging hardware) and even outperforms them in some cases (for instance, `hhhhhhhh` for CSMT merging hardware and `llllllll` for SMT merging hardware) for the real memory model. Thus, using HMT improves the performance of a pure CSMT/SMT machine significantly with a lower merging hardware cost.

6.2 Performance Evaluation of Thread Selection Schemes

Finally, we present an evaluation of the effect of the different thread selection schemes described earlier in Section 4, namely static, FNB, Eq and Eq+FNB. Figure 13 shows the IPC obtained by different thread selection schemes for a CSMT merging hardware for both perfect and real memory models. Figure 14 shows the same data for a SMT merging hardware. The figures also include the performance obtained by pure CSMT/SMT machines ("CSMT 2:2", "CSMT 4:4", "SMT 2:2" and "CSMT 4:4") and the peak performance achievable ("SMT 8:8" and "CSMT 8:8"). For clarity, only the average IPC achieved for all workloads is shown in the figures. The detailed set of results are available in [7] and are omitted from the paper for space reasons.

In general, static thread selection has the lowest performance across all the schemes, while scheme Eq+FNB performs the best. On an average, for the real memory model, the Eq+FNB scheme with a CSMT merging hardware obtains a performance improvement of 38% over pure CSMT with a 2-thread merging hardware ("HCSMT 2:8 Eq+FNB" vs "CSMT 2:2") and an 18% performance improvement with a 4-thread merging hardware ("HCSMT 4:8 Eq+FNB" vs "CSMT 4:4"). With SMT merging hardware, Eq+FNB has a performance improvement of 48% with a 2-thread merging hardware ("HSMT 2:8 Eq+FNB" vs "SMT 2:2") and 29% with a 4-thread merging hardware ("HSMT 4:8 Eq+FNB" vs "SMT 4:4"). Further, the performance achieved by Eq+FNB with a 4-thread merging hardware is quite close to the peak performance achievable (within 2.4% for CSMT and 6.3% for SMT merging hardware).

Note that even though static thread selection is the lowest performing scheme, it still improves the performance significantly. On an average, with the real memory model, static selection has 29% higher performance than the pure CSMT machine with a 2-thread merging hardware ("HCSMT 2:8 Static" vs "CSMT 2:2"), and 11% with a 4-thread merging hardware ("HCSMT 4:8 Static" vs "CSMT 4:4"). With the SMT merging hardware, static selection has 31% higher performance than the pure SMT machine with a 2-thread merging hardware ("HSMT 2:8 Static" vs "SMT 2:2"), and 11% with a 4-thread merging hardware ("HSMT 4:8 Static" vs "SMT 4:4).

Another interesting thing to note is that for a CSMT merging hardware, Eq scheme outperforms FNB scheme (though the difference in performance is not much). This happens because the replacement thread selected by Eq scheme does not share the renaming value with other threads in the selection set. Thus, more instructions are in general issued simultaneously because of less resource conflicts. Even the best performing scheme, Eq+FNB, has only a very small performance advantage over Eq (1.6% with a 2-thread merging hardware and only 0.8% with a 4-thread merging hardware). Thus, Eq scheme seems to be the most suitable thread selection scheme with a CSMT merging hardware as it has a lower hardware complexity (similar to static) with a performance similar to the most complex Eq+FNB scheme. With SMT merging hardware, FNB scheme outperforms Eq in the real memory model (but the performance difference is small, on an average 2.4% with a 2-thread and 4.6% with a 4-thread merging hardware). This is opposite to the behavior observed with the CSMT merging hardware, where Eq outperforms FNB most of the time. This happens because CSMT merges instructions at cluster level, and thus it heavily depends on cluster renaming to reduce contention for clusters. On the other hand, SMT merging hardware can merge instructions from different threads even if they use same clusters. Hence, cluster renaming is not so critical for SMT merging hardware as for CSMT. As a result, replacing a blocked thread with a non-blocked one is more important than only checking the equivalent thread in SMT merging hardware. With perfect memory, where all the latencies are short, Eq scheme outperforms FNB, as the replacement threads selected by Eq fare better at merging because all selected threads have different renaming values.

In conclusion, while Eq scheme is not the best performer among the dynamic thread selection schemes, its performance is very competitive with a significantly lower hardware complexity. Hence, Eq is the most suitable dynamic thread selection scheme.

7. CONCLUSIONS

Several multithreading schemes like Interleaved MultiThreading (IMT) and Simultaneous MultiThreading (SMT) have been proposed to reduce the resource underutilization in

VLIW processors. IMT provides significant performance improvements if the number of threads supported is small because of its ability to reduce vertical waste. With a larger number of threads, less opportunities exist for removing vertical waste, resulting in only marginal performance improvements with IMT. On the other hand, SMT performance keeps on improving significantly because of its ability to also reduce horizontal waste by merging instructions from different threads. However, in SMT it is difficult to support even a small number of threads because of the complexity of merging hardware. In contrast, IMT does not require a merging hardware and can support a larger number of threads.

In this paper, we have presented Hybrid MultiThreading (HMT), a technique which combines the advantages of both IMT and SMT. HMT supports a larger number of threads than SMT with a given merging hardware cost. This is achieved by merging only a subset of threads at a time. Every cycle, a new subset of threads is selected. HMT is independent of the merging scheme used by the merging hardware. In particular, this paper evaluated HMT with operation-level and cluster-level simultaneous multithreading (SMT and CSMT). The paper also evaluated several thread selection schemes that are used to select the subset of threads to consider for merging every cycle namely static, equivalent (Eq), first non-block (FNB), and a combination of equivalent and first non-block (Eq+FNB).

The experimental results show that HMT significantly improves the performance over a pure SMT/CSMT processor. Even with the simplest static thread selection, there is a significant performance improvement of 31% with a 2-thread merging hardware and 11% with a 4-thread merging hardware over pure SMT. While with a more complex thread selection scheme like Eq+FNB, using HMT improves performance by 48% with a 2-thread merging hardware and 29% with a 4-thread merging hardware over pure SMT. Further, using HMT with a 4-thread merging hardware achieves a performance similar to an 8-thread merging hardware without having to incur the cost of 8-thread merging hardware. Also, the Eq scheme performs quite well even though it has a much lower complexity compared to FNB and Eq+FNB. Interestingly, Eq outperforms FNB with CSMT merging hardware but the contrary is true with SMT merging hardware. This arises because cluster renaming is not so critical for SMT merging hardware as for CSMT. Nevertheless, the performance difference is quite small, making Eq scheme the most attractive choice for the thread selection scheme, as it has a performance close to the most complex Eq+FNB scheme but a complexity similar to the simplest static thread selection scheme.

8. REFERENCES

[1] Colorspace Conversion Program Used in High Performance Printers, Personal Communication.

[2] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. J. Smith. The Tera computer system. In *ICS*, 1990.

[3] D. Barretta, W. Fornaciari, M. Sami, and D. Bagni. Multithreaded Extension to Multicluster VLIW Processors for Embedded Applications. In *DATE*, pages 748–749, 2005.

[4] P. Faraboschi, G. Brown, J. A. Fisher, G. Desoli, and F. Homewood. Lx: A Technology Platform for Customizable VLIW Embedded Processing. In *ISCA*, pages 203–213, 2000.

[5] M. Fillo, S. Keckler, W. Dally, N. Carter, A. Chang, Y. Gurevich, and W. Lee. The M-Machine multicomputer. In *MICRO*, 1995.

[6] J. A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Trans. Computers*, 30(7):478–490, 1981.

[7] M. Gupta, F. Sánchez, and J. Llosa. Hybrid Multithreading for VLIW Processors. Technical Report UPC-DAC-RR-CAP-2009-19.

[8] M. Gupta, F. Sánchez, and J. Llosa. Cluster-Level Simultaneous MultiThreading for VLIW Processors. In *ICCD*, 2007.

[9] M. Gupta, F. Sánchez, and J. Llosa. Merge Logic for Clustered Multithreaded VLIW Processors. In *EUROMICRO Conf. on Digital System Design*, 2007.

[10] J. L. Henning. SPEC CPU2000: Measuring CPU Performance in the New Millennium. *IEEE Computer*, 33(7):28–35, 2000.

[11] F. Homewood and P. Faraboschi. ST200: A VLIW Architecture for Media-Oriented Applications. *Microprocessor Forum*, 2000.

[12] J. Hoogerbrugge and A. Terechko. A multithreaded multicore system for embedded media processing. *Transactions on HiPEAC*, 3(2), 2008.

[13] Inverse discrete cosine transform, taken from ffmpeg. http://ffmpeg.org. last consult april 2008.

[14] R. Kumar, N. Jouppi, and D. Tullsen. Conjoined-Core Chip Multiprocessing. In *MICRO*, 2004.

[15] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *MICRO*, pages 330–335, 1997.

[16] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O'Donnell, and J. C. Ruttenberg. The Multiflow Trace Scheduling Compiler. *The Journal of Supercomputing*, 7(1-2):51–142, 1993.

[17] S. Rixner, W. J. Dally, B. Khailany, P. R. Mattson, U. J. Kapasi, and J. D. Owens. Register Organization for Media Processing. In *HPCA*, pages 375–386, 2000.

[18] N. Seshan. High VelociTI Processing. *IEEE Signal Processing Magazine*, 15(2):86–101, March 1998.

[19] B. J. Smith. Architecture and Applications of the HEP Multiprocessor Computer System. In *SPIE*, pages 241–248, 1981.

[20] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *ISCA*, 1995.

[21] E. Tune, R. Kumar, D. Tullsen, and B. Calder. Balanced Multithreading: Increasing Throughput via a Low Cost Multithreading Hierarchy. In *MICRO*, 2004.

[22] VEX Toolchain. www.hpl.hp.com/downloads/vex/.

[23] W.-D. Weber and A. Gupta. Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: preliminary results. In *ISCA*, 1989.

[24] x264 - a free h264/avc encoder. www.videolan.org/developers/x264.html. Last consult April 2008.