

Software Multi-Threading Report

Vasileios Porpodas ^{1 2}

supervisor: Francky Catthoor ²

¹VLSI Design Laboratory, University of Patras, Greece

²IMEC vzw, Leuven, Belgium

October 2008

Abstract This paper examines software multithreading, a technique enabling usual single-threaded processors to run instructions from different threads within one thread. The concept behind the technique is similar to the hardware multithreading, namely exploiting thread-level parallelism (TLP) and efficiently transforming it into instruction level parallelism (ILP). The technique is a source to source transformation, taking place just before the compilation sequence. It is an enabling step for richer and larger regions in the lower compilation phases. Our results show that the technique significantly increases the ILP of the application by producing much denser schedules compared to the initial threads, thus enhancing performance. Since it does not require any hardware support, it can be applied on every existing processor micro-architecture, but we mainly focus on wide issue VLIW processors since it is them who suffer most from poor ILP applications.

1 Introduction

Applications usually suffer from poor ILP which is caused by high-latency instructions, slow memory accesses and poor instruction parallelism within one code region. It causes big holes to be formed in the schedule and many wasted cycles at run-time, all of which degrade performance significantly. It is therefore quite common for a wide-issue processor to be constantly under-utilized.

It is a fact that the majority of processors in the market are designed to run one thread at a time. Thus, the instructions the processor executes at a certain moment in time come from one certain context. This seems good enough unless we consider the applications themselves, which can consist of many different processes or threads which usually are independent and can run in parallel. This inherent application parallelism is left unexploited by all but the multi-processors or the processors with hardware multithreading support. These architectures use this parallelism to fill-up the otherwise unused processor resources. The concept behind this is to exploit the available thread-level parallelism (TLP) by converting it into ILP. There are several flavors of it with the most flexible one being Simultaneous Multithreading (SMT) [9] in which each thread context can issue instructions to each and every slot (functional unit) on the processor. This increases the utilization of the processor resources (higher ILP), thus leading to better performance.

Our work is an attempt to provide most of the benefits of a hardware multi-threaded processor with none of the hardware cost. We call the proposed technique software multi-threading (sw-smt) (as opposed to the hardware based one (hw-smt)) (Fig. 1) and we show that it can apply to any existing processor

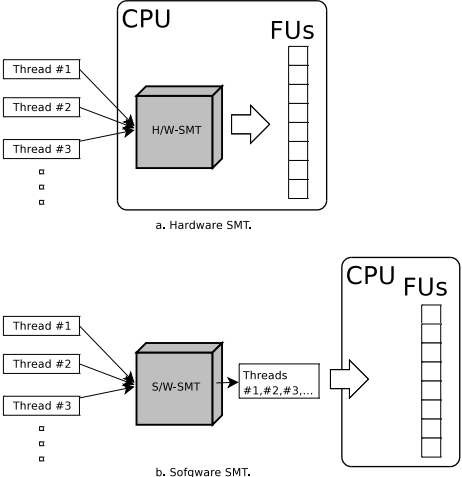


Figure 1: a. Hardware SMT: The thread mixing takes place at run-time on a hardware unit. b. Software SMT: The thread mixing takes place at compile time and the resulting code runs on a standard processor.

micro-architecture, independently of the number of contexts it can handle in parallel. The technique is most beneficial on wide-issue architectures which can provide the parallelism required support the high ILP generated by the concurrent thread contexts.

In more detail, the method followed is a source to source transformation of the threaded code. It involves identifying the code segments on which the sw-smt can operate on, rewriting the source code to remove any memory aliasing problems which could harm the TLP to ILP conversion if they came up later on and finally doing the appropriate code transformations to mix the source code of the two threads in such a way that later on, at the compilation stage, the TLP is visible and exploitable by the compiler. All of the above, of course, without sacrificing the application correctness.

The binary produced can run on any processor (although different optimizations can be triggered on different processors) and the resulting code contains instructions from different threads which can now run on any single-threaded processor in parallel with increased performance.

2 Related Work

The concept of software multi-threading is not a new one. [6] describes the main concept where each thread is analyzed using the thread-frames/nodes model [8]. Our approach follows the same concept and illustrates an actual technique which implements software multi-threading. The background related work comes from three domains where code merging takes place to increase the available paral-

lelism. The first one is the micro-architecture domain where the TLP is used as a means to fill-up the processor resources. The second one is the compiler domain, where automated compiler techniques expand the scope of the scheduling beyond the basic block. Finally there are some code transformations (mostly non-automated) which merge code from different threads.

In the micro-architecture domain there are several approaches towards TLP exploitation. The more straight forward one is the Chip MultiProcessor (CMP). Each thread or process can execute on a different processor of the CMP and therefore can execute in parallel with the others. The problem is that this approach is not the most efficient since each of the processors of the CMP is not fully utilized. This problem is addressed in [9] and it is described as vertical and horizontal waste. These refer to empty bubbles in the instruction schedule caused by long latency issues (such as long latency instructions or memory misses) and by insufficient ILP to fill up all the functional units at a certain cycle respectively. A more fine-grained approach is multithreading with its flavors each of which remove a certain kind of waste. The most flexible type of multithreading is Simultaneous Multithreading [9] which removes both vertical and horizontal waste by filling up all the available issue slots with instructions originating from the threads which are in flight. The concept of SMT implemented on a VLIW micro-architecture is discussed in [4]. However, this hardware based approach of exploiting the TLP is not always feasible due to its increased implementation cost and its higher energy budget, constraints which are quite common on mobile devices. The proposed technique, Software SMT, has similar result as the hardware SMT; the processor issue slots are filled-up with instructions from the other threads. An illustration of the instructions schedule of two threads and the sw-smt code for these threads is in Fig.2.

In the compiler domain we can find many automated techniques which improve the code parallelism by combining instructions from different parts of the program. One of the well known ones is superblocking [1] which forms a large region structure that enables the optimizer and scheduler to exploit ILP across basic block boundaries. A further optimization of this is the hyperblock structure [3] which is similar to superblock but tuned for predicated execution. However these techniques still have a limited scope compared to the source code techniques and as such they are still insufficient.

In the source code domain there are numerous attempts of moving and merging code. The most similar to our work is procedure cloning [7] where procedures are fused together to create a new denser hybrid procedure. The actual merging is done with a combination of standard loop transformations (fusion, peeling, splitting...) and other standard code motion transformations (replication and others). In our work we present a different more efficient and more generic way to merge looped or recursive codes and we specifically guide this process by focusing on manipulating the code regions formed by the compiler. Our method also exploits the increased performance offered by predication enabled processors.

The compiler domain has several techniques of merging code from different blocks in order to enhance the instruction parallelism in general. The most common technique in use by the compilers is the advanced region formation which can have various different implementations with the most popular ones being superblock and hyperblock [2]. The difference between these region formation algorithms and the basic block formation is that the latter is limited to con-

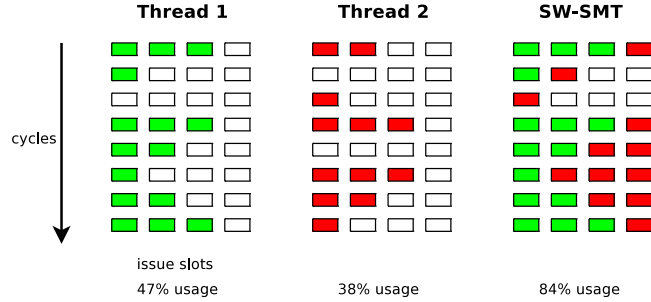


Figure 2: Instruction schedule for two threads independently and the resulting sw-smted code. The schedule becomes denser and the %FU usage is increased

tinuous code with no control instructions whereas the first ones can span over several control instructions. This shows that the regions formed can be much richer in terms of ILP than in the basic block case, thus leading to increased application performance. Our work takes into account how the standard region formation algorithms work when the source code transformations take place, so that the regions formed contain instructions from all the threads.

Finally in the source to source transformation domain one can view several attempts on merging independent code segments.

3 Methodology

The methodology consists of the following steps: i)Identify the critical parts of the threads ii)Check the schedule of the critical parts and decide if there is some space for increased ILP iii)Rewrite each thread to remove any memory aliasing issues which could come up after thread mixing iv)Merge the threads with a technique that ensures the creation of rich regions which contain instructions from the mixed threads. A detailed explanation of these steps follows.

The Identification of the critical parts can be done with a profiling tool. In our experiments we used the profiling figures produced by the coffee framework [5]. This step is quite important at this stage since the methodology is not automated and therefore we can focus only on a small fragment of the application. The typical case is that the critical part corresponds to a small code segment inside the loop with the most iterations. The loop acts like a barrier against the formation of large regions since the region forming algorithms [1],[3] consider only instructions from within a loop and not across several loops. We therefore have to do specific code modifications before handing over the merged code to the compiler; this is the job of step iv.

The second step (searching for the missing ILP) consists of passing each thread through the compilation sequence, and getting the final instruction schedule which gives us the Functional Unit (FU) utilization per cycle. A poor FU utilization can be caused by highly dependent code or a dependent chain of high latency instructions. Either way, such code is a good candidate for sw-smt. A bad candidate is some code with either completely packed schedule or one with lots of register file spills (spill code is easily tracked down).

Next, we have to make sure that no memory aliasing problems appear after

the thread mixing. Generally speaking, transforming a single-threaded code to a correct multi-threaded one requires special attention; Several data-types may need to be private inside each thread and doing so can be done in various ways. A quick and elegant, code style-wise, solution for this could be to use an array separated in parts, one part for each thread, so that data-types of each thread are mapped to a separate part of array. This ensures correctness and is semantically correct, but is not acceptable performance-wise, and is really inefficient when we try to perform sw-smt on such code. The problem is that this solution creates memory aliasing issues on most compilers, meaning that the compiler cannot automatically distinguish which parts of the array are uniquely accessed by each thread (it "thinks" that one thread could access data-types of some other thread), thus largely over-estimating the instruction dependences and therefore limiting the parallelization of the code. This will cause the sw-smt code achieve equal or lower performance compared to the single threaded one. A more "quick and dirty" way but one that actually works, is to give each data-type a different identifier (name). This removes any memory aliasing problems and enables further code optimization with sw-smt.

The final phase is the actual merging of the threads. As already mentioned, the critical part of each thread is usually a sequence of instructions within a loop body. This fact causes the formed region to be limited to instructions from a single thread only since no implemented region formation algorithm in existing compilers spans over different loops. We therefore had to come up with a source code transformations which would enable this much larger and richer code region. Our technique uses the control flow of one thread as a host code to embed inside it the critical sections of the remaining threads - we refer to them as guest code (Fig.3). The host code acts as a container for the guest code. The control condition depending on the profiling information can either be left unchanged or expanded to cover the majority of the host and guest conditions. Correctness is ensured by inserting the guard instruction just before the guest code. The guard is an if statement with the same condition as the condition of the loop in the original guest code. The resulting code is a single loop containing instructions from all the merged threads. The guard statements usually don't have an impact on the performance since the code after the guard is predicated, which is commonly supported in hardware by many processors. Therefore the guest code (the code after the guard) which is in the same code region as the code before it, can execute truly in parallel with the host code. If we could take a snap-shot of the processor execution, we would see instructions from many threads (host and guests) executed at the same time on the different processor functional units. After the merged code, for each guest code which is not guaranteed to complete its iterations, some tail code is added (which actually is a copy of the original looped guest code).

Recursive codes (such as the one we had) can be treated in two ways: i.remove recursion and apply the technique described above ii.do sw-smt following a recursive-only technique, which is what we will describe in this section.

Recursive functions consist of a function which, in its body, calls itself. We make the assumption that there exists some computation-worthy code before the call. This is the code are able to merge. If the call does not depend on the computation before it, we have the trivial case as the control moves smoothly from one call to the next one. The more interesting case is when the call is data-dependent on the computation before the call - this was the case in our

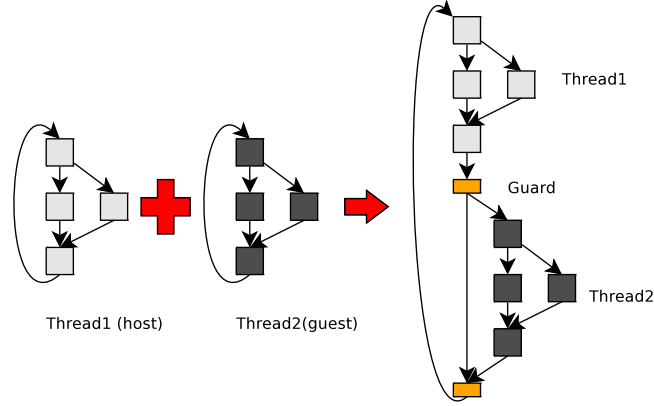


Figure 3: Thread Mixing technique for two threads (Control Flow Graph is shown). The guest loop is placed inside the host loop and is protected by some guard code

benchmark. The technique we propose is to create multiple functions each of them with different computation parts; one with all the computations from all the threads, one with all but one, etc., up until the functions just like the original ones with computations from only one thread. In our case we tried this for up to 3 threads in parallel and we ended-up with 2 functions more than the original ones (one for 3 threads and one for 2). The section of the code where the call was, is changed into a case code to enable calling of the different functions - each one of which with different amount of parallelism. The result is that depending on the status of the computation, we can exploit as much parallelism as possible by calling the appropriate function.

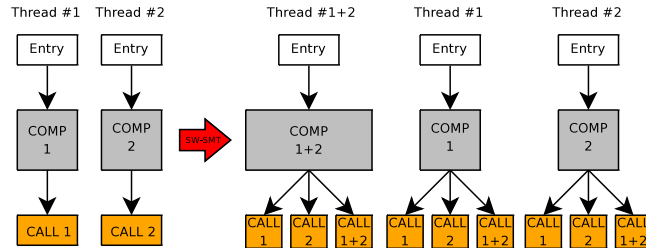


Figure 4: SW-SMT for two recursive functions. The function with a merged computation section is if possible

4 Simulation - Results

The benchmark we use is a maximum-likelihood decoding algorithm taken from the wireless domain called sphere-decoder [10]. This application is similar to a depth-first tree search and as such is highly dynamic. The simulator used for the exploration is coffee framework [5], a tool for architecture exploration.

Since in the wireless context the sphere decoder is executed on each and every

reception, it makes sense to run several of these in parallel. We experimented with sw-smt on various code versions of the sphere decoder. The first step was to remove the recursion from the original sphere-decoder code to ended-up with a much simpler and sw-smt-friendly loop code. We then applied the sw-smt technique as described in the methodology to weld together several threads. For our experiments, we applied sw-smt to mix from two up to five threads. We then ran (on the simulator) the resulting codes on a 8-issue VLIW processor and compared the cycles to running the threads sequentially on the same processor. The results are shown on Fig.5.

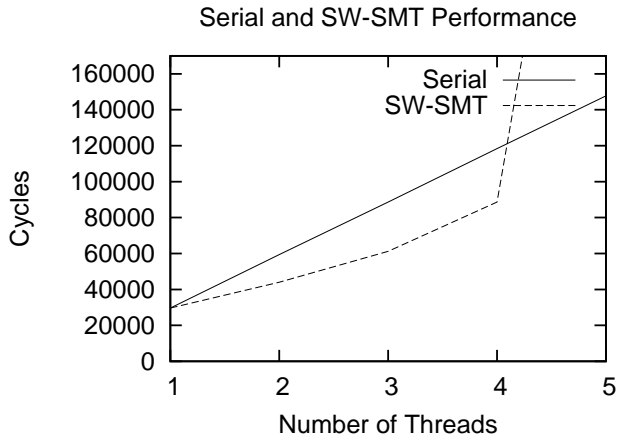


Figure 5: Sphere decoder performance with/without software smt on same data inputs

We also tested the efficiency of the sw-smt depending on the percentage of dynamic overlap of two threads 6. This gives us a rough estimate of the minimum and maximum gains one can get depending on the dynamism of the application sw-smt is applied on. The graph shows that depending on the application's input data, the overlap of the two threads can vary. This variation results also in an almost linear variation of the performance - with more overlap we get more performance , with no overlap sw-smt has no impact on performance.

The next step was to apply sw-smt on the recursive code. The difficulty in that is that it does not fit exactly the sw-smt technique for the non-recursive case because each thread can save its status through the recursive call independently and continue from where it left, which is something that can't be done if we use only one recursive function as a host with guest code in it. We therefore followed the approach for recursive functions described in the methodology section. In the case of 2 threads, we create 2 recursive functions; One double threaded (sw-smted) and one single threaded. Then at run-time the program decides which one of the two functions to call at every recursive call. We tested this technique for up to 3 threads on the sphere decoder benchmark.

The experiment was done, as previously, running the different sw-smted versions versus cases the one thread at-a-time cases. Each of the thread we ran had different inputs so the cycles of each one varied as shown in Fig.7. The results are illustrated on Fig.8.

The results show that in both cases the sw-smt technique does provide the

Serial and SW-SMT Performance depending on dynamic overlap

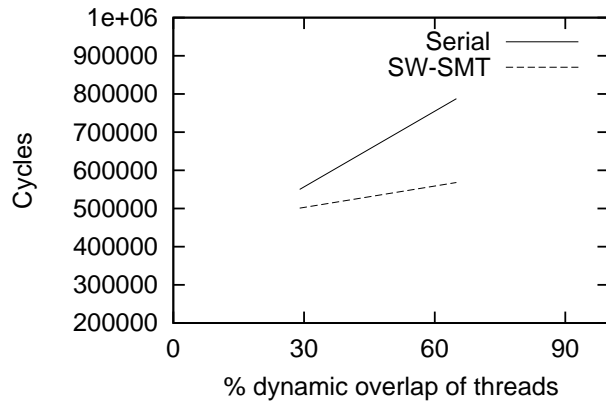


Figure 6: Impact of thread dynamism on Performance

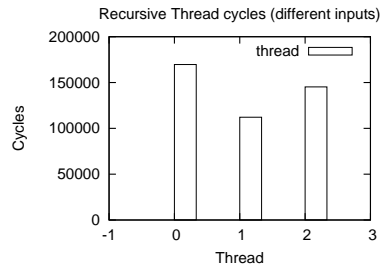


Figure 7: Recursive Thread Cycles. Each Thread has different input.

Serial and SW-SMT Performance on recursive Sphere Decoder with different data inputs

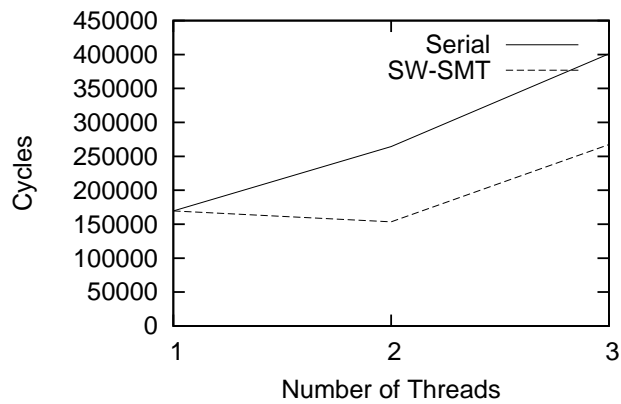


Figure 8: Recursive Sphere decoder performance with/without software smt on different data inputs

anticipated increase in performance. As we can see the sw-smt technique scales pretty well up until a certain point where the code is overwhelmed by register file spills.

5 Conclusion

In this paper we demonstrated sw-smt, a novel technique for running multiple threads in parallel, on processors which are designed to run only one thread at a time. The results show that in the case of a very dynamic code with very limited ILP, like the one tested, the application performance on a wide VLIW can increase significantly.

References

- [1] W.M.W. Hwu, S.A. Mahlke, W.Y. Chen, P.P. Chang, N.J. Warter, R.A. Bringmann, R.G. Ouellette, R.E. Hank, T. Kiyohara, G.E. Haab, et al. The superblock: An effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing*, 7(1):229–248, 1993.
- [2] Cliff Young Joseph A.Fisher, Paolo Faraboschi. *Embedded Computing - A Vliw Approach To Architecture, Compilers And Tools*. Morgan Kaufmann, 2005.
- [3] S.A. Mahlke, D.C. Lin, W.Y. Chen, R.E. Hank, and R.A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th annual international symposium on Microarchitecture*, pages 45–54. IEEE Computer Society Press Los Alamitos, CA, USA, 1992.
- [4] E. Ozer, T.M. Conte, and S. Sharma. Weld: A Multithreading Technique towards Latency-Tolerant VLIW Processors. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 192–203, 2001.
- [5] P. Raghavan, A. Lambrechts, J. Absar, M. Jayapala, F. Catthoor, and D. Verkest. Coffee: COMpiler Framework for Energy-Aware Exploration. *LECTURE NOTES IN COMPUTER SCIENCE*, 4917:193, 2008.
- [6] D.P. Scarpazza, P. Raghavan, D. Novo, F. Catthoor, and D. Verkest. Software Simultaneous Multi-Threading, a Technique to Exploit Task-Level Parallelism to Improve Instruction-and Data-Level Parallelism. *LECTURE NOTES IN COMPUTER SCIENCE*, 4148:12, 2006.
- [7] W. So and A. Dean. Procedure cloning and integration for converting parallelism from coarse to fine grain. In *Proceedings of the Seventh Workshop on Interaction between Compilers and Computer Architectures*, page 27. IEEE Computer Society Washington, DC, USA, 2003.
- [8] F. Thoen and F. Catthoor. *Modeling, Verification, and Exploration of Task-Level Concurrency of Real-Time Embedded Systems*. Kluwer Academic Publishers Norwell, MA, USA, 2000.

- [9] D.M. Tullsen, S.J. Eggers, and H.M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. *Computer Architecture, 1995. Proceedings. 22nd Annual International Symposium on*, pages 392–403, Jun 1995.
- [10] E. Viterbo and J. Boutros. A universal lattice code decoder for fading channels. *Information Theory, IEEE Transactions on*, 45(5):1639–1642, 1999.