Parallel Architectures
and
Compilation Techniques

PACT 2001

Work-in-Progress Session

Barcelona, Spain
September 2001

Editors:
Bruce Childers
Martin Schulz
Sally McKee

Foreword

This year's Parallel Architectures and Compilation Techniques (PACT 2001) includes for the first time a Work-In-Progress session. We received many excellent short abstract – some of which were wild ideas. From these we chose seven abstracts for presentation which are included in this booklet. We selected these abstracts for their novelty and the brevity with which these were expressed.

Thanks to Kevin Skadron, TCCA Newsletter Editor, we are also pleased to announce that all of the accepted and presented abstracts will appear in one of the next TCCA issues. This booklet therefore is only an intermediate presentation of the work for the conference attendees and is intended to form a better basis for a discussion of the presented work during the Work-in-Progress session.

We hope that the session itself will be lively, fun, and far from boring for both the presenters and the attendees and will serve as a forum for intensive, perhaps even controversial discussions. Keeping with the spirit of this session, there will be some interesting prizes for the best abstracts which will be selected by a panel of judges.

Many people contributed to the organization of this WIP. Thanks go to

- general chairs
- PC chairs
- Edu
- Marta for printing
- The students that will help run it (Sally do you have names?).
- submitters

We hope that this WiP session will be a success and that it will become a long and productive tradition for future PACT conferences.

Bruce Childers
Dept. of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
USA
childers@cs.pitt.edu

Martin Schulz
LRR-TUM (SAB)
TU München
80290 München
Germany
schulzm@in.tum.de

Sally A. McKee
Dept. of Computer Science
University of Utah
Salt Lake City, UT 84112
USA
sam@cs.utah.edu

Included Abstracts

A scheme for Selective Squash and Re-issue for Single-Sided Branch Hammocks
Karthik Sankaranarayanan, Kevin Skadron
University of Virginia, USA

Dynamic Way Allocation for High Performance, Low Power Caches
Matthew Ziegler, Adam Spanberger, Ganesh Pai, Mircea Stan, Kevin Skadron
University of Virginia, USA

Contrail Processors for Converting High-Performance into Energy-Efficiency
Toshinori Sato, Itsujiro Arita
Kyushu Institute of Technology, Japan

Predicting Trace Inputs with Dynamic Trace Memoization: Determining Speedup Upper Bounds
Maurício L. Pilla, Philippe O. A. Navaux, Federal University of Rio Grande do Sul, Brazil
Amarildo T. da Costa, Military Institute of Engineering, Brazil
Felipe M. G. Franca, Federal University of Rio de Janeiro, Brazil

Fuzzy Memoization for Floating Point Multimedia Applications
Carlos Álvarez, Jesús Corbal, Esther Salamí, Mateo Valero
UPC Barcelona, Spain

Communication Reduction for Scientific Applications on Shared-memory Multiprocessors
Arndt Balzer
University of the Federal Armed Forces Munich, Germany

Levo: IPC in the 10's via Resource Flow Computing
Augustus Uhl, David Morano, Alireza Khalafi, Marcos de Alba, Maryam Ashouei, David Kaeli
Northeastern University, USA
Sean Langford, University of Rhode Island, USA

A Scheme for Selective Squash and Re-issue for Single-Sided Branch Hammocks

Karthik Sankaranarayanan, Kevin Skadron
Dept. of Computer Science
University of Virginia
Charlottesville, VA 22904

Introduction

This abstract describes work to minimize re-execution of control independent instructions. This technique differs from prior work in its emphasis on compiler scheduling in order to minimize changes to the hardware of an out-of-order processor. Work so far has focused on single-sided branch hammocks.

A *branch hammock* [1] is an instruction sequence corresponding to an 'if' language construct. It is called *double-sided* when it corresponds to an 'if-then-else' construct and *single-sided* when it corresponds to an 'if-then' construct alone (*i.e.*, without the 'else' part). When the 'then' and the 'else' contexts contain only one basic block each, the branch hammock is called *simple*. In typical speculative processors, when a conditional branch corresponding to such a branch hammock is mispredicted, all the instructions fetched after the branch are *squashed*. However, the 'join' context is executed regardless of the direction of the branch. When a misprediction is detected, if the fetch engine of the processor went past the 'join' context, it fetched some potentially useful instructions too. If those instructions can be identified by some co-operative work between the compiler and the hardware, redundant re-fetching and re-execution can be eliminated. This work attempts to implement and evaluate such a co-operative mechanism in the particular case of *single-sided, simple branch hammocks.*

Related Work

Rotenberg *et al.* [2] have analyzed control independence in superscalar processors. Their paper analyzes the bounds of potential performance improvement due to the exploitation of control independence and assesses the complexity of possible implementations. Sodani *et al.* [3] have done a detailed study on dynamic instruction reuse. The potential *branch hammock* reuse described above, called squash reuse, is a subset of such dynamic instruction reuse. However, the re-fetching of instructions is not eliminated in their technique, and reuse techniques typically require substantial, multi-ported lookup tables and other hardware support. Rychlik *et al.* [4], proposed the reduction of value misprediction penalties by re-issuing of value-mispredicted instructions to the functional units, thus eliminating their re-fetching and re-renaming. However, their work does not examine re-issue with respect to branch mis-speculation. Klauser *et al.* [5] proposed the dynamic predication technique for reducing misprediction penalties in case of simple branch hammocks. The instructions of a non-predicated instruction set are predicated dynamically using hardware augmentation.

Overview of the Implementation

This work combines compiler scheduling with ideas from the above mentioned works (control independence, re-use, re-issue) in the domain of optimizing *single-sided, simple branch hammocks.* In such hammocks, the join context is control independent of the branch and hence need not be re-fetched on a misprediction. The execution of the instructions in the 'join'context that are data independent of the 'then' context is not erroneous and hence these instructions can be safely re-used. However, the execution and the dependency information of the 'join' context instructions that *are* data dependent on the 'then' context is erroneous. Hence, they have to be re-issued after the misprediction recovery in order to receive the proper operand values. For the above branch hammock scenario, when there is co-operation from the compiler, minimal addition to the hardware can implement such *selective squashing* (the 'join' context is not squashed) and *selective re-issue* (the dependent instructions should be re-issued).

In our work so far, the scheduler identifies the *single-sided simple branch hammocks* in a program. It finds the instructions in the 'join' context that are data independent of the 'then' context and groups them together at the beginning of the 'join' context. The code motion takes place in such a way that no anti or output dependences are violated. It then annotates the branch instruction with the size of the hammock, the number of independent instructions and the offset of the join context. With such 'grouping support' from the scheduler, the hardware implementation of selective squash and re-issue can be done with minimal cost. Consider an out-of-order processor model with an in-order commit stage. The following figure (Fig. 1) shows a simple branch hammock in the instruction-reorder buffer as a mispredicted branch is about to be discovered.

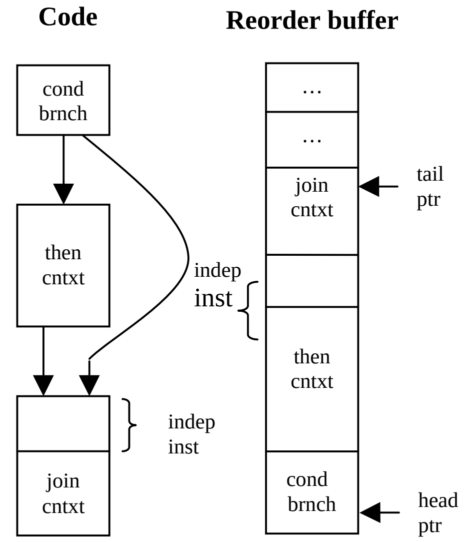


Fig. 1: State of the instruction-reorder buffer containing a hammock and a mispredicted branch.

	comp ress	gcc	go	ijpeg	li	m88k sim	perl	vor tex
Scheduled hammocks	5	984	341	68	13	85	40	13
Max. hammock size	4	8	7	9	4	7	6	9
Max. then context size	3	6	6	3	2	4	3	3
Max. no. of indep. Inst	2	6	4	6	1	3	4	1
Avg. hammock size	4	4	3.3	4	3.4	3.9	4	5.2
Avg. then context size	2.4	2.2	2.2	2.3	2	2.4	2.1	2.2
Avg. no. of indep. Inst	1.4	1.4	1	1.5	1	1.2	1.7	1

Table 1: Static scheduling data

Selective squashing of the ‘then’ context is easily implemented in the hardware by just bringing the head pointer to the beginning of the ‘join’ context. The effects of the ‘then’ context instructions on the register rename map is repaired by logical masking operations of the current map with the register maps at the position of the branch and at the beginning of the join context. Also, the dependent instructions of the ‘join’ context should enter the rename stage once again. In order to achieve this as simply as possible, the implementation expands the instruction buffer between the fetch and decode stages and holds instructions there. The dependent instructions of a hammock are tagged in the instruction queue and are held there until the branch resolves, making this re-issue easy. On a misprediction, all instructions in the instruction queue except the tagged ones are squashed. This ensures that the dependent instructions get to re-issue to the functional units with the proper values. Also, when the mis-speculated branch was predicted taken, the processor remembers the start of the ‘join’ context in order to re-steer the fetch engine when necessary. Finally, on a successful branch resolution, all associated instructions are purged from the fetch queue. Together, these implementation features require minimal addition to hardware and should be feasible without affecting the clock rate. They also avoid large, dedicated, multi-ported re-use tables.

This work implements the features detailed above using the SimpleScalar v3.0 [6] simulator tool set and the PISA instruction set. Performance evaluation has been done by simulating SpecInt95 benchmarks on the modified simulator and then comparing with the unmodified version. For scheduling, the benchmarks were compiled to assembly using gcc 2.6.3 –O3, scheduled by the scheduler software, and then assembled into the binaries. These were the binaries run on the modified simulator. Data from the static scheduler is summarized in Table 1.

Initial Results and Future Work

Initial investigation of the results show that the candidates chosen by the scheduler are very small basic blocks of size 3-4 instructions, and the scheduler typically finds at most 1-2 independent instructions but often finds none. This result happens to be similar to success rates in filling branch delay slots [7], which makes sense because the scheduling task is essentially identical. The important difference here is that we are able to use scheduling to exploit control independence in wide-issue, out-of-order organizations and are not limited to a fixed delay-slot architecture.

Unfortunately, because single-sided hammocks expose so few independent instructions, the gains in IPC obtained with this technique are negligible, averaging less than 0.05%. Some benchmarks, like *compress*, have almost no suitable basic blocks (see Table 1). Dynamic predication [5], on the other hand, would provide better benefits in these scenarios. These results necessitate the extension of the scheduling technique to nested and double-sided hammocks also. An analysis of the effectiveness of predication in lieu of or in combination with such re-issue and re-use techniques should also be explored. A study of the contrasts and overlaps between the nature of control independence exploited by these techniques (predication, re-use/re-issue) is another interesting future direction. In addition to the immediate benefits from exploring control independence, it is also expected that this work will also guide efforts in using compiler analysis to directly improve branch-predictor performance and/or reduce predictor hardware requirements.

Acknowledgements

This material is based in part on work supported by the National Science Foundation under grant no. CCR-0082671.

References

[1] J. Ferrante, K. Ottenstein, and J. Warren. “The Program Dependence Graph and Its Use in Optimization”. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[2] E. Rotenberg, Q. Jacobson, J. Smith. “A Study of Control Independence in Superscalar Processors”. In *Proc. of the 5th International Symposium on High Performance Computer Architecture*, January 1999.

[3] A. Sodani and G. S. Sohi. “Dynamic Instruction Reuse”. In *Proc. of 24th Annual International Symposium on Computer Architecture*, pages 194–205, July 1997.

[4] B. Rychlik, J. Faistl, B. Krug, and J. P. Shen. “Efficacy and Performance Impact of Value Prediction”, In *Proc. of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, Paris, October 1998.

[5] A. Klauser, T. M. Austin, D. Grunwald, B. Calder. “ Dynamic Hammock Predication for Non-predicated Instruction Set Architectures”. In *Proc. of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, Paris, October 1998.

[6] D. Burger, T. M. Austin, "The SimpleScalar Tool Set, Version 2.0", University of Wisconsin-Madison Computer Sciences Department Technical Report #1342, June 1997.

[7] J. Hennessy and D. Patterson. *Computer Architecture: A Quanititative Approach*, 2nd ed. Morgan Kaufman, San Francisco, 1996.

Dynamic Way Allocation for High Performance, Low Power Caches

Matthew Ziegler, Adam Spanberger, Ganesh Pai, Mircea Stan, Kevin Skadron[†]
Departments of ECE and [†]Computer Science, University of Virginia, 22904
{ziegler, spanberger, gpai, mircea}@virginia.edu, skadron@cs.virginia.edu

1 Introduction

Current cache configurations are inflexible, in that they cannot be customized for the needs of individual programs. Concurrent processes often have negative effects on each other's cache entries. A further problem is that real-time applications that need guaranteed cache access times might not always get these guarantees in a multitasking environment. In this abstract, we propose a framework for customizing the cache configurations to individual programs using *dynamic way allocation*. We then describe cache configuration methodologies that permit dynamic way allocation.

2 The Proposed Reconfigurable Cache Model

Dynamic way allocation provides flexibility for favoring higher performance, lower energy dissipation, or any combination of the two, depending on the overall goals of the system. This technique treats the *ways* of a set-associative cache as discrete units that can be independently allocated. Dynamic way allocation supports such goals as scratchpad memory for DSP or real-time applications, stream buffering for media processing, load balancing for SMT paradigms, and reduced power for portable computing platforms. The model dynamically allocates ways to individual processes, similar to tinting columns, which has the benefits of cache reconfiguration for scratchpad memory, multitasking and stream processing [1]. However, our model maps each process to a set of ways allowing a unique configuration per process. The number of ways needed in the cache for a process is specified with a *way-request*. A process's cache configuration is determined via a programmable software routine, static compiler hints, or recalling previous configurations.

2.1 The Dynamic Way Mapper and the Way-Vector

A hardware unit called the *dynamic way mapper* (DWM) takes a way-request as an input and produces a *way-vector* based on the availability of ways and the current mappings of other processes. The approach is similar to [2], but at a higher level of granularity. More than one process can be mapped to the same way. The DWM attempts to efficiently match the cache configuration specified by a way-request to a set of physical ways. We include the concept of a priority process for processes that need guaranteed cache access times, *e.g.* scratchpad memory in DSP applications. The DWM simply locks ways assigned to priority processes, preventing their use in other way-vectors. This means that a process may not always get the requested way-vector when priority processes have been mapped. The DWM balances allocations so that way-vectors are evenly divided among the ways. Thus, it tries to maximize way utilization. Way balancing may be implemented by attaching a counter to each column and incrementing the counter whenever a new process is mapped to a column. The dynamic way mapper can read the counters and determine the most efficient mapping.

The way-vector represents a mask for ways that are accessible by a particular process. Initially, a process provides a way-request to the DWM producing a way-vector, which is stored in a register (having one bit per way). During a context-switch, the process saves its way-vector along with other state information. When the process regains control of the processor, the DWM attempts to restore the previously stored way-vector. If a priority process has locked ways, then the DWM restores the ways that are not locked, but common to the previous configuration. If this is not possible, it tries to allocate as many ways as requested in the way-request.

2.2 Reconfiguration Framework

Ideally, we want each program phase to have its optimal cache configuration. In our model there are three ways to configure the cache for phases in the program at runtime: static compiler-hinted reconfiguration, dynamic program profiling and a hybrid of the two. Static compiler-hinted reconfiguration exploits compiler-based profiling to identify phases and suggest optimal cache configurations. Dynamic program profiling uses a programmable software routine to identify when and how to reconfigure the cache. It also has the ability to record a trace of configurations for a program. In case the compiler can only identify phases but not optimal configurations, the hybrid configuration scheme uses the compiler hints to pinpoint program phases, which then triggers the software routine. Static compiler hints allow faster cache reconfiguration, as they do not incur the overhead of software-based configuration. These compiler hints can be as small as one instruction indicating the way-request for the corresponding program phase. However, many programs have no idea of available cache size at compile time, and neither target runtime environment nor program behavior can be determined at compile time. Until such a compiler is available, the software configuration routine provides an alternative.

The software routine overcomes the drawbacks of static reconfiguration. It also allows reconfiguration of caches for legacy code or code that cannot be recompiled. In order to be effective, the software routine may need to run frequently, and it must run without high overhead. While the software routine could run as a process causing a low-overhead context switch, we feel that it would be beneficial to have a small, dedicated register file allowing it to run without causing a full context switch. Programs often may have different cache access trends for different phases of a program, *e.g.*, the optimal cache configuration for a subroutine may differ from that for the main program body. The main body of the program may itself have phases that favor different cache configurations. Ways can be dynamically adjusted at runtime to accommodate the cache needs of program phases.

The flexibility of our cache model to target a variety of system goals, such as high performance or low power, stems from its ability to program the software-configurable allocation routine. A different routine can thus be designed to reconfigure the cache to a variety of system goals, like high performance and low power, with existing hardware. Albonesi *et al.* uses a *performance degradation threshold* to determine a desired cache configuration with appreciable results [3]. Our model uses common runtime statistics such as IPC, miss rate, and cache reference counts that performance counters already provide, plus additional parameters like operating temperature and power dissipation statistics to trigger reconfiguration. These statistics are held in dedicated registers, and are accessed by the software routine to determine when and how to reconfigure the cache.

For example, the software routine could be written to be energy aware. As one part of this, we would like to be able to control the number of active ways in the cache to manage static power. We can use the available runtime statistics to design a power metric that accounts for both dynamic and static power consumption. Dynamic power could be measured as the number of cache misses times the estimated power consumed by each cache miss. Static power could be measured as the instruction count times the number of active ways times the static power consumed by a way per instruction (which is a function of operating temperature). We could then write

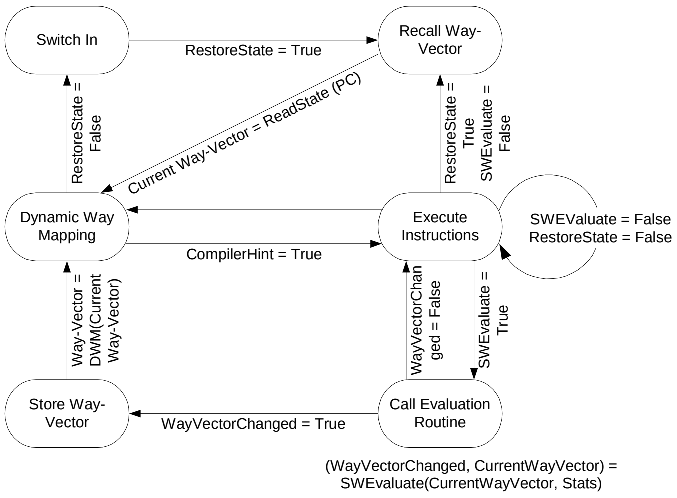


Fig.1: Reconfiguration operations of the proposed cache architecture

the software configuration routine to minimize the power metric either by adding more active ways (to reduce dynamic power from cache misses) or by reducing the number of active ways, thus reducing static power. The flexibility of the software configuration routine allows configuration metrics and controls to be as simple or as complex as the system designer desires.

Fig.1 shows the reconfiguration operation from the time a process switches in. Our approach permits us to build a trace of possible cache configurations for a process and then use the trace when the process is next switched in.

2.3 Configuration Recall

If a program uses dynamic reconfiguration via the software routine, it is useful to store the way-vectors for various phases of the program. We include a mechanism called the *way-history table* (WHT) in our cache model to record the optimal configurations for the different phases and then recall them at a future time. The WHT stores the PC and the way-vector of a process at certain phases of the program. It is useful to only store way-vectors that are optimal, and therefore we erase way-vectors that are not performing well. We reduce the size of the WHT by storing only differential way-vectors *i.e.*, the PC and way-vector that start a new program phase. Since the storage overhead is low we may store the WHT to memory when the process loses context, allowing it to be recalled when the process regains context. We extend this concept by storing the WHT to memory with the program. At this point we reduce overhead further by storing only way-requests instead of way-vectors. This can be viewed as an attempt to customize a cache to an individual program.

2.4 Low Power Modes

Turning off portions of the cache helps curb current leakage in unused columns, thus reducing static power dissipation, and it lowers dynamic power dissipation by reducing the number of ways requiring pre-charging. The effectiveness of turning off columns, for appreciable power savings, has been demonstrated in [4]. Our concept provides the means to tradeoff performance and power depending on the chosen system goals. The significance of static power dissipation increases as transistor gate lengths are reduced, *i.e.*, as technology scales. However, if the goal is an overall energy savings for the system, it is important that the amount of static energy saved be more than the increase in dynamic energy caused by the smaller cache. In many cases, a smaller cache will cause an increased number of cache misses, resulting in increased dynamic energy dissipation. Since energy can be measured as the power times execution time, a smaller cache may cause execution time to increase, which may cause the total energy to increase. We must consider these secondary effects of reducing cache size when attempting to minimize energy. Further, static power is exponentially dependent on the temperature. Thus, an accurate estimate of static power versus the secondary effects of reducing cache size cannot be made unless the system temperature is monitored.

2.5 Full-Way Power Down vs. Partial-Way Power Down

There are two techniques to reduce static power consumption. One is powering down a way or a number of ways. However, this poses a problem when we want to power down a portion of the D-cache holding dirty data (*configuration consistency*). The dirty columns in the cache cannot be immediately shut down because information will be lost. We may either write back all the dirty cache lines immediately or write-back some lines while transferring some others to an active portion of the cache. Regardless of which method is used, we need a mechanism to determine when the portion in question has been "cleansed". This may be expensive and will create power overhead for turning off a cache portion. This is not a problem with the I-cache because the I-cache is never dirty. Another type of configuration consistency concerns both D and I-caches, and arises when a process has useful cache entries in columns not included in the column vector. There are two methods to handle this situation, *hard partitioning* and *lazy partitioning* [3], [5]. Hard partitioning only accesses the ways specified by the way-vector, by pre-charging the valid columns for a cache access. But the secondary effects of increased cache misses may negate the energy saved from less pre-charging. Lazy partitioning accesses all ways.

The second approach is to partially power down some ways by lowering the V_{dd}. This is similar to the dynamic voltage scaling techniques employed in the Crusoe processor by Transmeta. The technique avoids losing the data, but does not provide as much static power saving as the first low power method.

3 Simulation Methodology

We will use the Wattch Toolkit [6] for simulating our re-configurable cache model. We intend to emulate a multithreaded environment by modifying the toolkit. Running two or more benchmarks in our multithreaded environment, we will compare conventional caching techniques against our proposed model. The Wattch framework will allow us to observe the performance and power characteristics of the proposed cache model.

4 Concluding Remarks

The proposed reconfigurable cache model will allow cache customization on a program-to-program level. The model profiles program execution, identifies optimal cache configurations and reconfigures the cache accordingly. The reconfiguration framework permits configuration to be based on global system goals, trading off performance and power for a desired optimal combination.

References

[1] D. Chiou, L. Rudolph, S. Devadas, *et al.*, "Dynamic Cache Partitioning via Columnization", *Computation Structures Group Memo 430*, MIT

[2] B. Nayfeh, Y. Khalidi, US Patent 5584014, "Apparatus and method to preserve data in a set-associative memory device", Dec. 1996

[3] D. H. Albonesi, "Selective Cache Ways: On-Demand Cache Resource Allocation", *Journal of Instruction-Level Parallelism*, May 2000.

[4] Michael Powell *et al.*, "Gated-V_{dd}: A Circuit Technique to reduce leakage in Deep-Submicron Cache Memories", In *Proceedings of the 2000 International Symposium. on Low Power Electronics and Design*, July 2000.

[5] P. Ranganathan, S. Adve, N. Jouppi, "Reconfigurable Caches and their Application to Media Processing", In *Proceedings of the 27th International Symposium on Computer Architecture,* June 2000.

[6] D. Brooks, V. Tiwari, M. Martonosi, "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations", In *Proceedings of the 27th International Symposium on Computer Architecture*, June 2000.

Contrail Processors for Converting High-Performance into Energy-Efficiency

Toshinori Sato[1,2] Itsujiro Arita[1]
[1] Department of Artificial Intelligence
[2] Center for Microelectronic Systems
Kyushu Institute of Technology
{tsato,arita}@ai.kyutech.ac.jp

1 Introduction

Current trend of increasing popularity of portable and mobile computer platforms such as notebook PCs and smart cell phones is a driving force to investigate high-performance and energy-efficient microprocessors. For example, Java-2 MicroEdition (J2ME) works on cell phones. We can download a game and play it on our cell phone. Travellers guide and flight ticket reservation are available. Furthermore, mobile banking and trading are also provided. As computing power of mobile device increases, its energy efficiency becomes a first class design constraint. It is important to note that energy consumption is more important for mobile devices than power consumption because it decides the lifetime of batteries.

The energy consumed in a microprocessor is the product of its active power and execution time. Thus, to reduce energy consumption, we should decrease both of or either of them. The active power P_{active} and gate delay t_{pd} of a CMOS circuit are given by

$$P_{active} = fC_{load}V_{dd}^2 \tag{1}$$

$$t_{pd} \propto \frac{V_{dd}}{(V_{dd} - V_{th})^{\alpha}} \tag{2}$$

where f is clock frequency, C_{load} is load capacitance, V_{dd} is supply voltage, and V_{th} is threshold voltage of the device. α is a factor depending upon the carrier velocity saturation and is about 1.3–1.5 in advanced MOSFETs [3]. Based on Eq.(1), it is easily found that power supply reduction is the most effective way to lower power consumption. However, Eq.(2) tells us that supply voltage reduction increases gate delay, resulting in slower clock frequency. And thus, computing performance of microprocessor is diminished.

In order to mitigate the performance loss, we can exploit parallelism [1]. Two identical circuits are used in order to make each unit to work at half the original frequency while the original throughput is maintained. Since the speed requirement for the circuit becomes half, the supply voltage can be decreased. In this case, the amount of parallelism can be increased to further reduce the total power consumption. In this paper, we propose to utilize another kind of parallelism, which is thread level parallelism, for energy reduction with maintaining processor performance.

2 Contrail Processor Architecture

To reduce the energy consumption, we divide an execution of an application into two streams [6]. One is called *speculation stream* and consists of the main part of the execution. However, it exploits trace level value prediction [5], and thus several regions of the execution are skipped. In other words, the number of instructions in the speculation stream is smaller than that in the original execution, resulting in energy reduction. In contrast, the other stream is called *verification stream* and supports the speculation stream by verifying each data prediction. The key idea is that the verification stream can execute slowly if the data prediction accuracy is considerably high. We can reduce the clock frequency of the datapath for the verification stream. Furthermore, the supply voltage is also reduced. From these consideration, its energy consumption is significantly reduced.

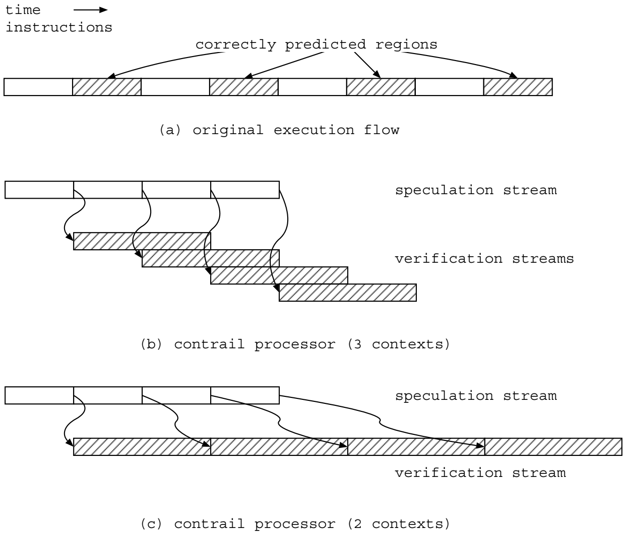
Each stream executes as a thread on a simultaneous multi-threading processor [2, 8], whose execution core consists of dual speed pipelines. The speculation stream is dispatched into a high-speed pipeline (speculation pipeline) and the verification stream is dispatched into a low-speed and low-supply-voltage pipeline (verification pipeline). In the ideal case, that means there are no misspredictions, the speculation stream finishes silently and waits for the verification process. In the case where a missprediction occurs, the execution of the speculation stream is squashed at the point where the missprediction is detected and processor state is recovered by the verification stream.

We call this *Contrail Processor Architecture.* What is a contrail ? A contrail is the condensation trail that is left behind by a passing jet plane. The speculation stream runs ahead just like a jet plain, and the verification stream is left behind by the speculation stream and fades away just like a contrail. One of the differences from the previously proposed pre-computing architectures [4, 7] is that the contrail processor architecture does not rely on redundant execution. In the ideal case, the number of instructions executed is unchanged. Another is that its target is improving energy efficiency instead of improving performance.

The potential of the contrail processor architecture on energy-efficiency is estimated as follows. We decide that clock frequency and supply voltage for the verification pipeline are half of those for the speculation pipeline. Assuming that half of the original execution of an application is ideally predicted and is distributed uniformly as explained in Figure 1(a), the execution is divided into the speculation and verification streams on a contrail processor with three contexts (1 and 2 for the speculation and verification streams respectively) as depicted in Figure 1(b). The predicted regions are skipped in the speculation stream and execute in the verification streams with enlarging their execution time. Energy consumption is calculated as follows. For the speculation stream, it becomes half of the original execution since the number of instructions reduced by half. In contrast, for the verification streams, the sum of every execution time remains unchanged since the execution time of each instruction increases by double while the total number of instructions reduced by half. Its energy consumption is decreased by the reduction of the clock frequency and the supply voltage. Based on Eq.(1), it is reduced to $\frac{1}{8}$. Thus, total energy savings is 37.5%. In addition, the application gains higher performance. On the other hand, when the contrail processor has only two contexts, the execution time becomes slightly longer as explained in Figure 1(c). In this model, every thread constructing the verification stream is kept in a FIFO queue when it can not obtain its dedicated context. It is true that the effectiveness of the contrail processors (energy savings) depends on the value prediction accuracy and the size of each predicted region. However, we have confirmed that the potential of the contrail processors on energy saving is substantial.

3 Summary

In this paper, we proposed an energy-efficient processor architecture based on value prediction and multi-threading techniques. These techniques exploit thread level parallelism, resulting in mitigating performance loss due to the supply voltage reduction. From preliminary estimation, the proposed architecture has a potential of approximately 40% energy savings with maintaining processor performance.



(a) original execution flow

(b) contrail processor (3 contexts)

(c) contrail processor (2 contexts)

Figure 1: Execution on a contrail processor

Acknowledgments

This work is a joint study with Semiconductor Technology Academic Research Center (STARC).

References

[1] A.P.Chandrakasan and R.W.Brodersen, “Minimizing power consumption in digital CMOS circuits,” Proc. of IEEE, vol.83, no.4, April 1995.

[2] J.Emer, “Simultaneous multithreading: multiplying Alpha’s performance”, Microprocessor Forum, October 1999.

[3] T.Hiramoto and M.Takamiya, “Low power and low voltage MOSFETs with variable threshold voltage controlled by back-bias”, IEICE Trans. on Electronics, vol.E83-C, no.2, February 2000.

[4] A.Roth and G.Sohi, “Speculative data-driven multithreading”, 7th Int. Symp. on High-Performance Computer Architecture, Junuary 2001.

[5] R.Sathe, K.Wang, and M.Franklin, “Techniques for performing highly accurate data value prediction”, Microprocessors and Microsystems, vol.22, no.6, November 1998.

[6] T.Sato, “Study on application of high-performance processor architectures for energy-efficiency”, Proposal for STARC joint research program, September 2000 (in Japanese).

[7] K.Sundaramoorthy, Z.Purser, and E.Rotenberg, “Slipstream processors: improving both performance and fault tolerance” 9th Int. Conf. on Architectural Support for Programming Languages and Operating Systems, November 2000.

[8] D.M.Tullsen, S.J.Eggers, and H.M.Levy, “Simultaneous multithreading: maximizing on-chip parallelism”, 22nd Int. Symp. on Computer Architecture, June 1995.

Predicting Trace Inputs with Dynamic Trace Memoization: Determining Speedup Upper Bounds

Maurício L. Pilla[1], Amarildo T. da Costa[2],
Felipe M. G. França[3], Philippe O. A. Navaux[1]
[1] Informatics Institute, Federal University of Rio Grande do Sul, Brazil
{pilla, navaux}@inf.ufrgs.br
[2] Military Institute of Engineering, Brazil
amarildo@cos.ufrj.br
[3] COPPE, Federal University of Rio de Janeiro, Brazil
felipe@cos.ufrj.br

Abstract—

This paper proposes the combination of Dynamic Trace Memoization (DTM), a trace reuse technique, with value prediction techniques. By combining both kinds of techniques, we show that it is possible: (i) to reuse traces when all source values are ready and matching stored values, or (ii) to predict values based on partially ready source values matching stored values of a reusable trace. The potential of the new composite scheme is demonstrated by comparing DTM results obtained with and without perfect prediction. Results obtained with perfect value prediction showed a large potential of performance increase, about 22% over the base architecture (HM over the SpecInt95 benchmark set).

Keywords— Superscalar Architectures, Trace Reuse, Value Prediction

I. Introduction

Dynamic Trace Memoization (DTM) [1, 2] is a trace reuse technique based on instruction and, most importantly, trace memoization. In order to take advantage of redundancy and value locality found in dynamic executions, previously seen redundant instructions are stored in a Global Memoization Table (Memo_Table_G). When a sequence of redundant instructions is identified, a new entry is stored in a Trace Memoization Table (Memo_Table_T).

The reuse domain is composed by all integer instructions without collateral effects. Floating-point instructions are not reused because they show small redundancy. Only address calculation is reused in loads and stores.

Every instruction in the reuse domain is searched in both tables. If an entry is found in Memo_Table_T and its input scope matches the present register values, the trace beggining with that instruction is reused; if an entry is not found in Memo_Table_T but in Memo_Table_G, the instruction is reused. Trace reuse may cross branch boundaries and collapse true data dependencies in a single cycle, increasing system performance. More detail can be found in [1, 2].

Although DTM shows good overall performance improvements, most reusable traces, i.e., already stored in Memo_Table_T, cannot be reused because of unresolved source values when the reuse test is done. Under such circumstances, traces may hold some correct values but, as DTM is not speculative, only when all source values are matching current register values a trace can be correctly reused. Thus, the potential increase in performance is not exploited in these cases.

The present novel proposal uses Value Prediction (VP) [3] to increase the number of reused traces by predicting the not ready input values instead of isolated instructions. This approach does not increase hardly the hardware: the values to be predicted are stored in the trace input scope. Hardware to fix speculation is also present in all superscalar processors to deal with branch prediction.

When a memoized trace is found and not all input values are ready, some of its inputs may be predicted based on the previous values and the trace is then speculatively reused. If the values are found to be wrong, the instructions are re-executed with the right values.

This combined mechanism can be selective without needing an additional confidence mechanism: predictions may be restricted to some cases, as when only one value is missing and only one trace matches the remaining input values. If the other input values are correct, there is a better chance values will be correctly predicted than when isolated values are predicted. In the same way, the prediction of branches inside traces can be considered as being confident in our mechanism.

II. Simulation Methodology and Results

Eight benchmarks from SPECint95 were chosen, cc1, compress, go, ijpeg, li, m88ksim, perl, and vortex. All benchmarks were simulated to the end using train inputs, except for perl and vortex, which executed 300 million instructions. The DTM simulator is derived from SimpleScalar's sim-outorder, with a five stages pipeline and MIPS-I ISA. The DTM with perfect predictor [1] uses an oracle to predict traces with inputs that are not ready in the reuse test but are correct. The architecture configuration is the same as explained in [1].

A. Performance

Figure 1(a) [1] shows the IPC with the base architecture, the DTM architecture, and the DTM architecture with perfect VP. The third columns present the results obtained with perfect VP of traces whose operands are correctly stored in traces but are not ready in the reuse test. These traces are the main target of our work. All benchmarks show significant increases in IPC with both DTM and DTM with perfect prediction.

Figure 1(b) [1] illustrates the speedups obtained over the base architecture. In this initial work, only traces with correct values are predicted. More sophisticated mechanisms, like stride prediction, are not considered now but they will be approached in our next works. The last two columns present the HM of speedups. There is a large potential of performance increase from 9.5% to 22.4% of speedup in DTM performance, an increase of 135% in speedup, with VP.

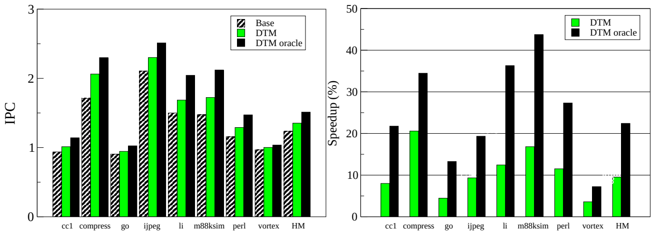


Figure 1. (a) IPCs for superscalar, DTM and DTM oracle architectures (b) DTM speedups over the base architecture

B. Traces Reused

Figure 2(a) shows the percent of reused traces and not reused traces because of unknown input values. The last column shows the HM. About 55% of the traces with valid inputs are not reused because of not ready input values. These traces may take advantage of VP to increase the overall performance. Using VP, the percent of reuse increase from 43% to 59% (Figure 2(b) [1], HM).

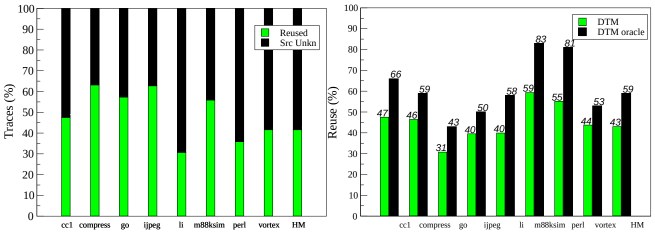


Figure 2. (a) Reused traces and traces with unknown source values (b) Trace and instruction reuse

III. Conclusions and Future Works

In this paper, we have presented a novel hybrid Trace Prediction-Reuse scheme joining Dynamic Trace Memoization and Value Prediction. The new mechanism should not require a significant increase in the hardware, using the resources already present. The potential of performance increase is very large, allowing an average speedup of 22.4% over the base architecture.

This proposal can be compared with storageless VP using prior register values [5] in the light of the small extra hardware needed, but we can also take advantage of possible correlations among input scope values to determine when predicting a value is a good choice. Instructions inside a trace, in this proposal, are not executed but reused. In this way, the waste of resources is minimized, and the potential results are substantially better.

Comparing with the trace-based predictor proposed in [4], our approach offers a potentially smaller hardware dedicated to prediction, as it does not need to execute the predicted trace (because it is reused), and it can be selective without the need of confidence hardware. Therefore, it can also minimize losts due to VP misspeculation.

We will be soon implementing a set of prediction techniques within DTM, and then we will analyze their impacts on performance and critical path execution. We also plan to predict values for instructions that generate inputs for traces to be able to speculatively antecipate trace reuse. Value prediction of these values has a great potential to collapse critical paths.

References

[1] A. T. da Costa. Exploiting the Reuse of Traces Dynamically at the Processor Architecture Level. PhD thesis, COPPE/UFRJ, Apr. 2001.

[2] A. T. da Costa, F. M. G. França, and E. M. Chaves Filho. The Dynamic Trace Memoization Reuse Technique. In Proc. of the International Conference on Parallel Architectures and Compilation Techniques, pages 92–99, Philadelphia, Oct. 2000.

[3] M. H. Lipasti and J. P. Shen. Exceeding the Dataflow Limit Via Value Prediction. In Proc. of the 29th Annual ACM/IEEE International Symposium on Microarchitecture, Paris, Dec. 1999.

[4] R. Sathe, K. Wang, and M. Franklin. Techniques for Performing Highly Accurate Data Value Prediction. Microprocessors and Microsystems, 22(6):303–313, Nov. 1998.

[5] D. M. Tullsen and J. S. Seng. Storageless Value Prediction Using Prior Register Values. In Proc. of the 26th Anual International Symposium on Computer Architecture, pages 270–281, Atlanta, May 1999.

Fuzzy Memoization for Floating Point Multimedia Applications

Carlos Álvarez, Jesús Corbal, Esther Salamí, Mateo Valero
Departament d'Arquitectura de Computadors. UPC. Barcelona
e-mail: {calvarez,jcorbal,esalami,mateo}@ac.upc.es

Multimedia applications are, indisputably, one of the driving forces of computer architecture today. Most embedded processor designs have evolved trying to overcome the handicaps of future media processing protocols and killer applications such as MPEG-4, highly-demanding 3D games, or applications targeted at the third generation of mobile digital phones (UMTS).

Current low-power embedded media systems still present several performance shortcomings. Most of the new media applications are migrating to floating-point computing due to the increase in complexity of the protocols and the media data types. Unfortunately, most of the media embedded systems today do not have floating point units (and thus, have to emulate them by software). The floating point units are expensive in terms of area and power, and several of the floating point instructions exhibit extremely high execution latencies that degrade overall performance.

Tolerance of media applications

Media applications has a unique characteristic which is not found in any other kind of application (neither scientific/numerical nor integer SPECint-style): media data outputs are extremely tolerant. By tolerant, we understand that they are highly robust to losses in their precision. Results from 3D, video, image or audio applications could present errors in the computation that would not be visually (or audibly) perceptible, and thus, the result of the execution would be considered correct.

Indeed, multimedia applications output data is directed to persons, not to computers. 'Human systems' do not care about exact results. In fact, this is the property used by image and audio compression systems (such as MPEG or MP3) to achieve high compression rates. In other words, they are exchanging accuracy by size. We propose to add another exchange trade-off consistent on trading accuracy for processing speed. In the world of embedded media processing there may be a lot of situations where speed (or power) is a factor almost as important as compression (e.g. for low/medium quality screens, in the presence of real time constraints or when having battery-life concerns).

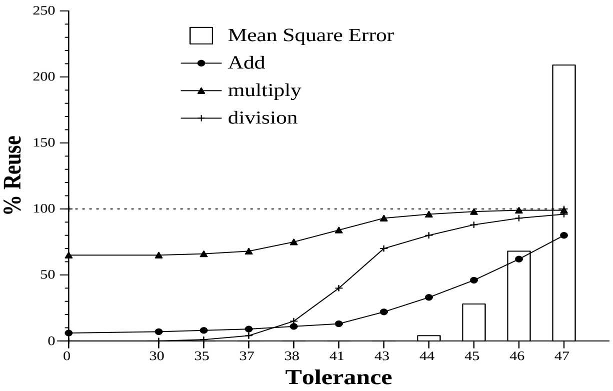


Figure 1. Potential reuse with added error for different tolerances.

Introducing tolerance into memoization

Some recent studies [1] have shown that memoization is a powerful technique to avoid computation in long latency instructions such as multiplication and division. Every time such an operation is invoked, their operands and their result are stored in a *Reuse Table*. When there is another instance of the same operation with the same inputs, then the computation can be avoided. This technique not only improves the performance of floating point computations but also produces important savings in energy consumption. The major drawback of the technique is the need to implement very large *reuse tables* to achieve acceptable reuse rates, which may provide diminishing returns from the point of view of power consumption and processor cycle time.

We are currently studying how tolerance can improve floating point instruction memoization, by trading accuracy by speed (and power efficiency). With conventional memoization, we need two exact instances of the same operation in order to obtain reuse. Our claim is that if we have two instances with similar inputs, the result will also be similar enough to the desired one to allow the second instance to be skipped by using the output of the first.

This method increases memoization reuse opportunities

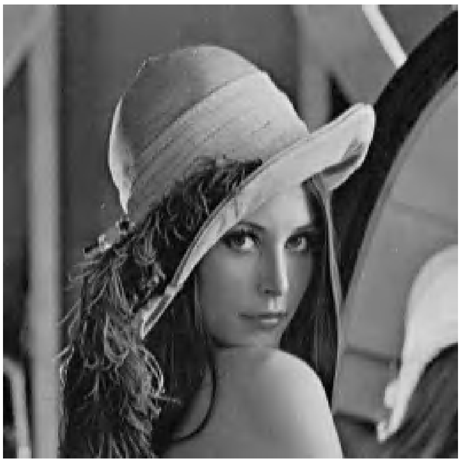


Figure 2. Image codified with original EPIC algorithm.

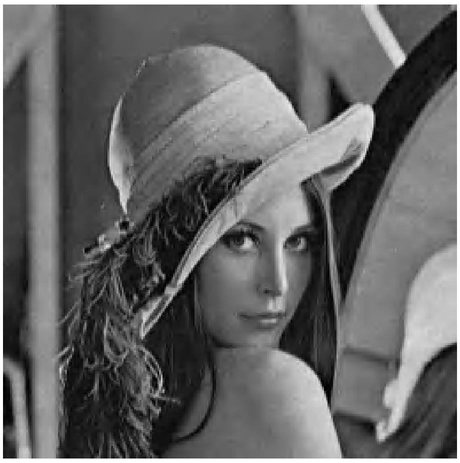


Figure 3. Image codified with EPIC algorithm with tolerant reuse (46 bits; 68% error)

at the cost of introducing some errors in the result. However, this error is not significant for the output due to two main reasons. First, floating point is generally used in multimedia for its range rather than for its precision. Therefore, we are typically overcomputing data (from the point of view of maintaining precisions that are lately discarded). Second, we are processing media data types (such as image pixels or audio fragments) that are extremely 'tolerant' by nature; that is, the human observer is not going to realize the error due to the filtering effect of his senses.

We have started our study with the EPIC image compression application from the *Mediabench* suite. We have measured the potential reuse of floating point instructions with different tolerance degrees. By tolerance degree we understand the number of bits from the mantissa (out of 52) to be ignored when testing a hit in the reuse table.

Figure 1 shows the percentage of reuse of different floating point operations for different degrees of tolerance, assuming unbounded tables. The figure includes also a metric of image quality degradation in the form of the *Mean Square Error* of the output image. Results show how reuse hit rates are poor when no tolerance is implemented. However, when tolerance is introduced, we may observe how reuse hit rates grows significantly with no appreciable image degradation up to a tolerance degree of 43.

The results obtained also show that with low errors there is a large working area for tolerance. Between 43 and 46 bits of tolerance, the error keeps low while the reuse grows to near a 90% in all the operations. In order to realize qualitatively the impact of the introduced error at those ranges, we may look at figures 2 and 3 where it is shown the original image and the image with a tolerance degree of 46 (that is, with a 68% of mean square error). As shown, even with a high MSE of a 68%, image quality is still high (although, evidently, differences exist).

We are currently developing a framework to better evaluate the impact of tolerant reuse for floating point operations in an embedded system model. We are confident that great benefits may be provided in terms of execution time, power consumption and energy efficiency due to the high reuse rates achieved by the fuzzy memoization paradigm.

References

[1] D. Citron, D. Feitelson, and L. Rudolph. Accelerating multi-media processing by implementing memoing in multiplication and division units. *ASPLOS VIII*, 1998.

Communication Reduction for Scientific Applications on Shared-memory Multiprocessors

Arndt Balzer
Department of Computer Science
University of the Federal Armed Forces Munich
85577 Neubiberg, Germany
balzer@informatik.unibw-muenchen.de

1. Introduction

Many numerical applications from the field of scientific computing can only be solved on parallel computers in a senseful way. An important class of applications are continuous problems that are discretised and solved with iterative numerical algorithms. While solving the problem all processing elements involved have to transmit large amounts of data to and from the shared-memory. Applications are therefore characterised by the communication to computation ratio. The total per-processor execution time can be divided into three categories: busy time in computation, time spent at synchronisation events and time waiting for remote data. Runtime analyses of such scientific applications have shown, that the time spent waiting for data is relevant to the total execution time [3]. Since communication in a shared-memory systems is usually demand-driven by the consuming processor, the problem is overlapping communication with computation. This is not a problem for small data items, but it can degrade performance if there is frequent communication or a large amount of data is exchanged [4]. Most scientific applications use the double-precision floating-point data format (fp). Using single- or double-precision operands in the ALU makes no difference to execution time. However it makes a big difference to the amount of data to be exchanged and therefore time spent on communication. The time considered in this short paper is the time waiting for data that cannot be overlapped, e. g. by using prefetch mechanisms. Our approach is to reduce the amount of communication data by a variable data format, discussed in the following section.

2. Data reduction approach

The idea behind the work is rather to transmit the bits of the operands as required for computation than the full operand as stored in memory. There are two mechanisms to reach this goal: On the one hand, data with maximum precision is provided but not always needed. Many times only the significant bits of the operands are required. E. g., during the initial phase of a calculation transmitting the sign, exponent and the first fraction bits will be sufficient. The less significant bits are irrelevant at that point of time. Therefore a mechanism for a dynamic precision for communication is useful.

On the other hand, when using iterative algorithms, data asymptotically converges towards a steady state. For example close to the stopping criterion only the less significant bits change from one iteration to the next. Writing back the already fixed bits to the global memory is redundant and causes unnecessary communication. A combination of both mechanisms leads to a maximum reduction of data communication because operands are only partially transmitted. Figure 1 shows how data is usually stored in memory. For simplification we assume that memory is arranged in words of 256 bits (smallest addressable word). As a consequence, some variables are implicitly grouped to a word. To efficiently benefit from the mechanisms the data in the global memory has to be rearranged. Without any changes bus-based systems with e. g. 64 or 128 bit busses can not make use of the mechanisms because one or two 64-bit-words are transmitted per cycle anyway, with or without redundant bits. To profit from the reduction described above words have to be grouped to logical blocks and these blocks have to be rotated. Figure 2 shows the operands stored "vertically". The effect is that parts of the operands become addressable. So the sign, the exponent and the fraction field or parts of it become selectable.

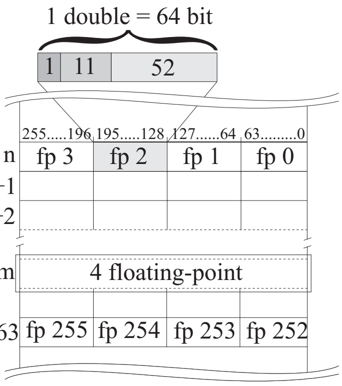


Figure 1. Variables conventionally stored in memory

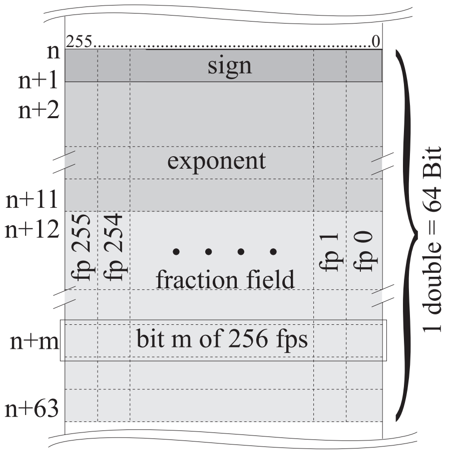


Figure 2. Variables rotated and arranged in blocks

3. Simulation environment

We use a testbed for a shared-memory architecture to study the execution behaviour of selected applications. The (physically) shared-memory is both used for communication and data storage. The testbed emulates a UMA architecture. We use the SPMD programming model with explicit communication. For coherence purposes hardware semaphores are used for shared data. We intentionally did not rewrite code to achieve good results. We just made changes where required (communication routines). When running the applications, we measure the application behaviour at different points of execution. For applications with repetitive steps, we measure only a few steps.

4. Experiments and first results

For validation some applications/kernels from popular benchmark suites have been adapted. We adapted *water* from the SPLASH [5], *FFT* from SPLASH 2 [6], *mg3* from NPB [1] and further on a *CFD* from FORTWIHR [2] and a proprietary *2D-Laplace* using SOR. The major data type of all programs is double-precision floating-point. They use iterative methods for solving and require an intense communication. From first results we can say:

- The amount of data to be exchanged can be reduced significantly. In our experiments the reduction of redundant data communication varies. In *water* and *FFT* it is approximately up to 25%, in *2D-Laplace*, *mg3* and *CFD* it is up to 50%. However, the impact to the total execution time depends on several factors like the application's communication behaviour and the problem size. So no general statement about the improved overall performance can be made.
- In Section 2 we said that several variables have to be grouped and therefore might unnecessarily be transmitted. We have found that this blocking-technique (to stay with our example: 256 fp-operands) is not disadvantageous. For example, in a 3D-CFD simulation any point on the grid is determined by a velocity vector with 3 components, one scalar for temperature and pressure each and further auxiliary variables. So only few grid-points fit into a single block. Problem sizes that get in conflict with such a small number of grid points would not be run on a multiprocessor (poor parallel efficiency).
- The memory latency increases because the operands cannot be copied into the local memory until the corresponding data block transfer is completed. On the other hand the effective bandwidth in the same way. When several data blocks are required the initial latency becomes more and more insignificant.
- Maximum performance is gained by applications using dynamic scheduling according to the frequently changed data region.

5. Conclusion and future work

In this paper we have presented an approach to reduce the communication overhead for certain types of applications using shared-memory systems. The initial results are promising. Redundant data can easily be found by comparing data already read with data to be written. Combined with a ”rotating unit” it can be performed automatically by the interface's hardware. The implementation does not cause any problems for today's hardware. Determine the currently required precision is more complicated. The dynamic precision is a user defined parameter in the subroutine call so far. We have to determine the reduced precision's impact on the computation in more detail: Does it (generally) cause more iterations? If it does, what is the time spent on additional computation in comparison to the time saved in reduced communication before? A concept for a hardware support for read-accesses is currently elaborated. In the case that the current precision is not applicable to all data, the management (by the programmer) might become complex. Referring to Figure 2, a hardware implemented vector in the shared-memory –like an instruction pointer– that points on the relevant data might be more comfortable than software variables.

References

[1] D. Bailey et al. The nas parallel benchmarks 2.0. *Report NAS-95-020*, 1995.

[2] M. Griebel et al. *Numerische Simulation in der Strömungsmechanik*. Vieweg, Braunschweig, 1995.

[3] D. Jiang and J. P. Singh. Scaling application performance on a cache-coherent multiprocessor. *Proc. of the 26th International Symposium on Computer Architecture*, 1999.

[4] D. E. Lenoski and W.-D. Weber. *Scalable Shared-Memory Multiprocessing*. Morgan Kaufmann Publishers, San Francisco, 1995.

[5] J. P. Singh, W.-D. Weber, and A. Gupta. Splash: Stanford parallel applications for shared-memory. *Computer Architecture News*, 20, no. 1, 1992.

[6] S. C. Woo et al. The splash-2 programs: Characterization and methodological considerations. *Proc. of the 22nd International Symposium on Computer Architecture*, 1995.

Levo:
IPC in the 10’s via Resource Flow Computing

Augustus Uht*, David Morano, Alireza Khalafi**, Marcos de Alba****
Maryam Ashouei, Sean Langford* and David Kaeli****
Northeastern University **
Boston, MA
University of Rhode Island *
Kingston, RI

1.0 Extended Abstract

The goal of our work is to produce a machine execution model that realizes much of the ILP (Instruction Level Parallelism) existent in general purpose programs, without requiring program recompilation. We look to obtain IPC (Instructions Per Cycle) in the 10’s, designing a system with processing capabilities that scale linearly (in terms of the number of ALUs)

A 1992 study by Lam and Wilson demonstrated there exists a lot of ILP in common programs [1]. More recent studies [2, 3, 4] have reinforced this earlier work. However, no one has yet created a machine that can begin to exploit even a small fraction of this parallelism. Current high-end superscalar processors do take advantage of ILP by having many instructions in flight at a given time, but the net resulting IPC (Instructions Per Cycle) is always low (in the low single digits).

Therefore the primary goals of our Levo machine are to simultaneously realize: 1) high IPC performance; 2) linear hardware scalability [5], and 3) a realizable design. We base our work largely on the ability to achieve these goals on legacy code (SPECint), without recompilation, so as to make the work as generally applicable as possible.

2.0 The Levo Solution: Resource Flow Computing

The Levo machine makes extensive use of data-flow and control-flow speculation; more so than Lipasti and Shen’s Superspeculative Machine[6]; more so than any machine we are aware of. Levo executes standard control-flow based programs using methods that go beyond both the standard control flow model and the data flow model.

Levo attempts to exploit as much speculation as possible; this is achieved via a *resource flow* computing model. With resource flow, processing elements are assigned to the highest priority instructions that have not been executed, regardless of whether their inputs or operands are known to be available (data flow constraints), and potentially speculatively (control flow constraints). The rest of the execution time is spent applying programmatic data flow and control flow constraints to the instructions in Levo's Execution Window, potentially squashing and re-executing a single instruction many times, so as to end up with programmatically-correct execution of the code.

2.1 Levo Operation and Microarchitecture

Levo consists of three major components: an Instruction Window, an Execution Window and a Memory Window. Levo operates as follows. In the Instruction Window, the Instruction Fetch unit uses branch, jump, and other predictors, as well as predication hardware, to buffer a large number of instructions loaded from the memory system in the Instruction Load Buffer. The Execution Window is arranged in a matrix fashion, with the Temporally Earliest Instruction (TEI) in the upper left-hand corner and the Temporally Latest Instruction (TLI) in the lower-right hand corner. The Execution Window is *n* rows by *m* columns of instructions. The left-most column holds the first n instructions, and so on. Thus the Instruction Load Buffer typically loads a whole column of instructions into the Execution Window at a time. This is easier done than might it seems, since the Execution Window is a static instruction window, that is, the instructions nominally appear in the window in the same order they exist in memory, which is independent of the actual control flow of the executing program.

In our model, we do not provide any centralized register file or renaming buffer; registers values are present only on buses and in bus buffering units. Each of the register buses is similar in design to the Tomasulo Common Data Bus, though Levo adds to this the time tag of the register data. In the Levo Memory Window, multiple versions of a single address co-exist. Uniqueness is maintained through the use of time tags. Figure 1 shows the internal organization of the Execution Window for the Levo machine.

3.0 Summary

We have modeled Levo at three different levels of abstract: 1) HDLevo: a VHDL model that will be targeted for an FPGA prototype system, 2) LevoSim: a cycle-accurate timer, and FastLevo: a trace-driven simulator. Running SPEC2000 codes, we have obtained IPC of 13-21 instructions per cycle with FastLevo on a configuration with 1024 processing elements, modeled with a perfect memory system and perfect branch prediction.

Legend

‘as’ - active station, registers only.

‘AS’ - complete Active Station.

‘p’ - Predicate Forwarding Unit.

‘mf’ - Memory Forwarding Unit.

‘rf’ - Register Forwarding Unit.

‘M’ - Mainline path.

‘D’ - Disjoint Eager Execution path (multipath).

‘PE’ - Processing Element.

‘TTCOL’ - column part of Time Tag.

Characters in *italics* - part of running example.

[Time Tag, Reg. Address, DST value] - bus tuple.

[, ,], [, ,] - temporally sequential bus tuples.

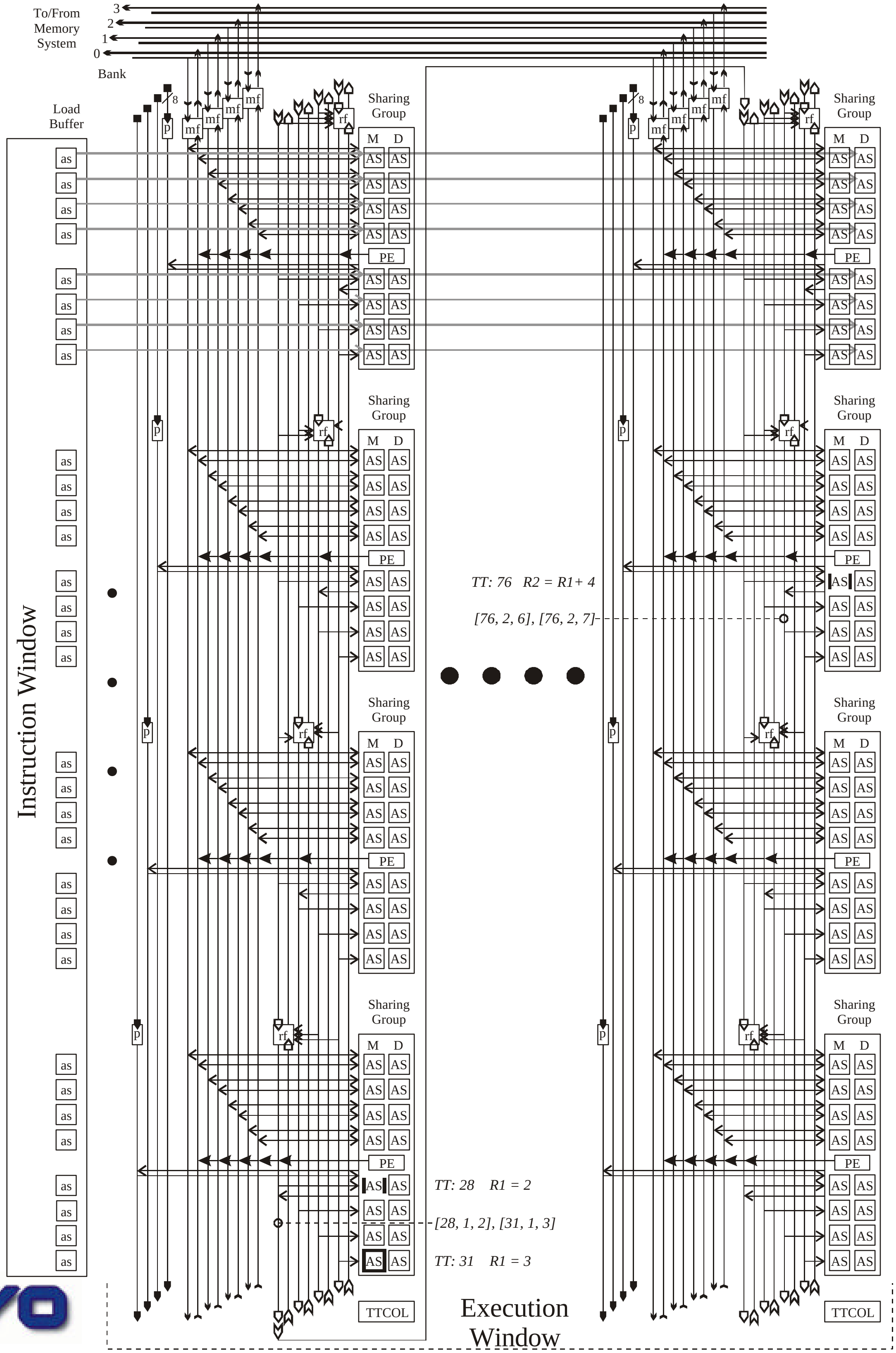


Figure 1. The Levo Execution Window

References

[1] M. S. Lam and R. P. Wilson, "Limits of Control Flow on Parallelism," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*. Gold Coast, Australia: IEEE and ACM, May 1992, pp. 46-57.

[2] J. G. Cleary, M. W. Pearson, and H. Kinawi, "The Architecture of an Optimistic CPU: The Warp Engine," in *Proceedings of the Hawaii International Conference on Systems Science (HICSS)*, vol. 1: University of Hawaii, January 1995, pp. 163-172.

[3] A. K. Uht, "A Theory of Reduced and Minimal Procedural Dependencies," *IEEE Transactions on Computers*, vol. 40, no. 6, pp. 681-692, June 1991.

[4] A. K. Uht and V. Sindagi, "Disjoint Eager Execution: An Optimal Form of Speculative Execution," in *Proceedings of the 28th International Symposium on Microarchitecture (MICRO-28)*. Ann Arbor, MI, November/December 1995, pp. 313-325.

[5] D. S. Henry, B. C. Kuszmaul, and V. Viswanath, "The Ultrascalar Processor: An Asymptotically Scalable Superscalar Microarchitecture," in *HIPC '98*, December 1998, URL: http://ee.yale.edu/papers/HIPC98-abstract.ps.gz.

[6] M. H. Lipasti and J. P. Shen, "Superspeculative Microarchitecture for Beyond AD 2000," *IEEE COMPUTER*, vol. 30, no. 9, pp. 59-66, September 1997.

[7] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal of Research and Development*, vol. 11, no. 1, pp. 25-33, January 1967.