Toward a Toolchain for Pipeline Parallel Programming on CMPs

John Giacomoni, Tipp Moseley, Graham Price, Brian Bushnell,
Manish Vachharajani, and Dirk Grunwald
University of Colorado at Boulder
{John.Giacomoni, moseleyt, Graham.Price, Brian.Bushnell, manishv, grunwald} @ colorado.edu

Abstract

Today's processors exploit the fine grain data parallelism that exists in many applications via ILP design, vector processing, and SIMD instructions. Thus, future gains must come from chip-multiprocessors, which present developers with previously unimaginable computing resources. Programmers can use these resources for coarse-grain data-parallel computation or task parallelism. Given the extensive research history in coarse-grain data parallelism, we argue that the right approach is to invest research effort on task parallelism because it is currently poorly supported in programming languages, operating systems and performance analysis tools. Such an approach encourages refactoring working sequential applications into task-parallel, and in particular pipeline-parallel, applications. Thus, we join the minority chorus that believes the best strategy for developing parallel programs may be to evolve them from sequential implementations.

There are challenges; future multi-core systems are likely to be heterogeneous and consist of many types of cores. Programmers need support in understanding and exploiting such systems. We believe that the systems community needs to focus on building complete toolchains that encompass all four stages of parallel program development for task parallelism: identification, implementation, verification, and runtime system support. This paper discusses this vision and our efforts in developing such a toolchain.

1 Introduction

Traditionally, increases in transistors and fabrication technology have led to increased performance. However, these techniques are showing diminishing returns due to limitations arising from power consumption, design complexity, and wire delays. In response, designers have turned to chip-multiprocessors (CMPs) that incorporate multiple cores on a single die. While CMPs are a boon to throughput driven applications such as web servers, single-threaded applications' performance remains stagnant. This is because the typical approach to parallelizing software (data-parallel or task-parallel) has been to find, extract, and run nearly independent code regions on separate processors [1]; a difficult task for general purpose applications [2].

An alternative and more promising approach is to use a special task-parallel organization called a pipeline-parallel organization. This is accomplished by decomposing a task into a series of sequential stages connected by a data-forwarding mechanism. Data-dependencies are easily handled, provided each datum only references previous data. Further, throughput may increase proportionally to the depth of the pipeline with a short completion interval. For these reasons, modern hardware systems, from microprocessors to routers, are built on a pipeline design. While software-based pipelines have been proposed in the past, only today's CMPs deliver the resources to capture the performance benefits of software pipeline-parallel organizations.

Note that the proposed approach requires exten-

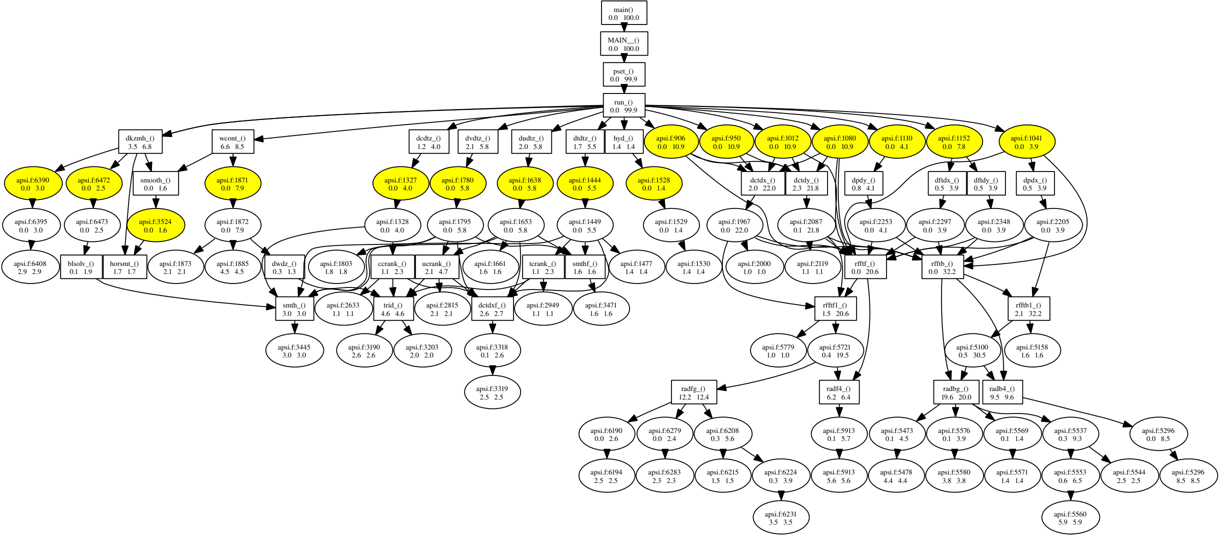


Figure 1: A partial loop/call graph for the top 1% of loops/functions in a sequential version of 301.apsi. Highlighted nodes were parallelized by the program's authors in the SPEC2001 OMP suite.

sive tools support. Without tools to identify pipeline-parallelism, implement pipeline constructs, and validate a pipelined application it will be difficult for this model to gain wide spread acceptance. Thus, we have been working on a new generation of tools that encompass the entire pipeline-parallel program development cycle. For identification, we have LoopSampler, LoopProf [3, 4], and ParaMeter [5]. LoopProf and LoopSampler permit developers to visualize the relationship between functions and hot loops in a loop/call-graph without recompilation and with negligible overhead. ParaMeter permits developers to visualize pipeline-parallelism and explore dependences on traces with over a billion instructions in only 1 GB of RAM. For implementation we have built the FastForward [6] software engine for supporting micro-and macro-scale pipeline-parallel constructs called Concurrent Threaded Pipelines [6]. These pipelines yield performance even in situations requiring communication to complete in less than a main-memory access (≈35ns per operation) while handling sequential inter-data dependencies. For verification we are developing a tool to support intelligent static analysis of programs, with the intent of characterizing their run-time behavior. This information will then allow the same tool to assist in test generation, verify portions of the program, perform post-mortem root-fault analyses, and apply performance optimizations. Additionally, we have begun developing a virtualization environment to abstract away the heterogeneous execution environments of the future that will mix general purpose processing cores with specialized execution units and even reconfigurable FPGA fabrics.

2 Work in Progress

Our work in progress discussion is organized as follows. First, we discuss our work on LoopProf, LoopSampler, and ParaMeter. Then we discuss our work with Concurrent Threaded Pipelines followed by our work on verification. We conclude with a discussion on the need for virtualized execution environments.

LoopProf and LoopSampler Given a sequential program or an algorithm, deciding where and how to parallelize the code is often tedious and time consuming. Traditional profilers are well suited to analyzing hot spots in code and increasing performance in inner loops, but effective thread-level parallelization requires coarse granularity that is not exposed by modern profiling tools (e.g., gprof).

LoopProf [3] and LoopSampler [4] generate a loop/call graph (see Figure 1). Loops are oval nodes

and functions rectangular. Each node presents the function's name or loop's filename:lineno. In addition, each node contains the percentage of self and total execution that it contributes. With this view of structure combined with an execution profile, deciding which loops to try to parallelize is greatly simplified. Those loops that reside high in the hierarchy and account for the largest percentage of total execution are the best targets. These loops may execute very few instructions themselves, but loops may be nested deeply within them.

LoopProf can generate much more detailed information than presented here. LoopSampler can generate the loop/call graphs with almost no overhead.

ParaMeter Unlike LoopProf and LoopSampler, ParaMeter is focused on providing developers with an interactive visualization and analysis tool to identify opportunities for pipeline parallelism through global analyses of fine-grain data-dependence structures in a large program trace. Figure 2 shows sample output from a ParaMeter prototype [7]. On the horizontal axis is the earliest time a particular instruction may execute, under ideal circumstances. On the vertical axis, we have the position of the instruction in the original trace (i.e., the dynamic instruction number). A line from lower-left to upper-right represents a tightly coupled dependence chain. Each of these dependence chains could be extracted as a pipeline stage. Nearby dependence chains are extracted as neighboring stages in the pipeline. Note that as one focuses in on a particular point in time each line will exhibit additional dependence chains and thus more threads. The bottom graph helps developers focus on important regions by showing the maximum instructions per cycle that are possible at each time step.

Interactive visualization and analysis of this graph requires rapid global analysis and random-access to billions of sequential instructions. If stored uncompressed, these traces can take terabytes of storage, making the required global analysis and random-access too slow for the interactivity required for the tools to be useful. We have developed a compression technique based on Binary Decision Diagrams [8] that permits developers to access over a billion instructions with only 1 GB of RAM [5]. Unlike previous techniques which require decompression for

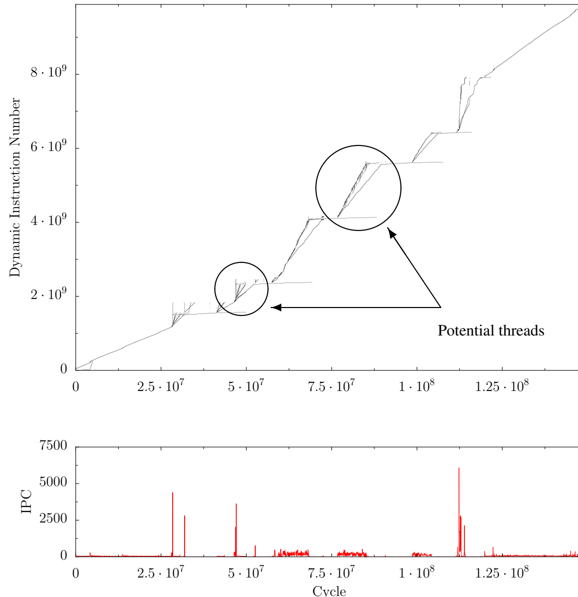
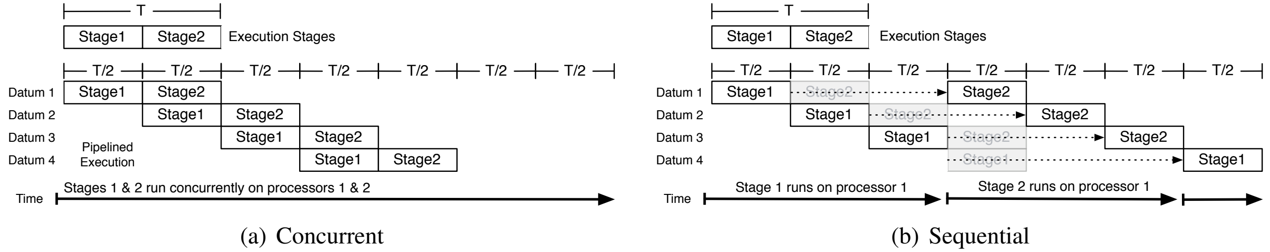


Figure 2: Dynamic instruction number vs. Ready-Time plot of SPEC CINT 2000 benchmark 254.gap. Circled areas represent potential threads.

access, our compression technique permits random-access and analysis *without decompression*.

Concurrent Threaded Pipelining Our implementation toolchain efforts are based on optimizing pipeline-parallel structures called Concurrent Threaded Pipelines (CTP) [6]. CTPs complement existing task- and data-parallel (e.g., parallelizing outer loops) techniques by permitting applications to easily handle sequential inter-data dependencies without synchronization. CTPs may improve throughput proportionally to the depth of the pipeline. Figure 3(a) depicts a two-stage pipeline demonstrating potential throughput improvements with pipeline-parallelism. Unfortunately, with insufficient dedicated computations resources it is impossible to reify performance improvements, Figure 3(b) depicts this. Examples of applications parallelizable with CTPs include video decoding, scientific computing, and network intrusion detection.

Concurrent Threaded Pipelines address the previous issues by requiring a very low-cost communication mechanism and that every stage of a pipeline to be concurrently scheduled. With FastForward [6],



(a) Concurrent

(b) Sequential

Figure 3: Pipeline Timing

software-only communication costs were reduced from 600-20,000ns (experimentally found) to ≈35ns per operation [6] using architecturally tuned concurrent lock-free queues. This permits CTPs to be used for a wide range of micro- and macro-scale optimizations that were previously not feasible. CTPs were used to build the Frame Shared memory (FShm) [9] architecture, permitting us to forward a record breaking 1.488 million frames per second (672ns per frame) in user-space with commodity hardware. Micro-scale optimizations that hide main memory latencies may also be possible, including Decoupled Access/Execute architectures [10] and DSWP [11].

Verification and Validation While our work on validation is in its early stages, it is a critical component given the difficulty of writing parallel programs. Current work is focused on identifying the frontier of program states that could lead to a previously observed failure (e.g., a core-dump or assertion failure). Note, however, that pipeline-parallel structures may be easier to validate given that the component-to-component interaction follows a well-known well-understoond pattern (i.e., stage to stage interconnections with localized pipeline state).

For the future work, unlike many automated approaches, we envision an interactive process where the user helps the verifier when it has difficulty. A promising approach may be to use formal methods to identify "fault lines" along which failures are likely, and then use guided test-generation to exercise code along these fault lines.

Virtualization General purpose platforms have begun to incorporate heterogeneous and reconfigurable hardware with wildly varying performance characteristics, capabilities, and programming models [12]. Leveraging all the resources while maintaining the general purpose nature of these processors will require a new virtualized system platform.

The FastForward [6] engine takes the initial steps by isolating pipeline stages on shared-memory systems with a portable communication engine. However, stronger virtualization will be necessary as systems begin to include specialized components not capable of sharing memory directly. We have begun investigating more generalized virtualization abstractions to ensure programmers do not have to reason about the specific number and microarchitecture of computational resources on a particular platform to ensure correct execution.

3 Conclusion

In conclusion, we reiterate our belief that the best long term strategy for developing parallel programs is to aid developers in extracting task parallelism, and in particular, pipeline parallelism, from applications. However, finding, implementing, and supporting pipeline-parallel applications on a CMP processor requires extensive software tool support in each of these areas. A number of researchers, including the authors, have made substantial strides towards this end.

References

[1] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August, "Decoupled software pipelining with the synchronization array," in *13th International Conference on Parallel Architecture and Compilation Techniques (PACT'04)*.

Los Alamitos, CA, USA: IEEE Computer Society, 2004, pp. 177–188.

[2] S. Amarasinghe, "Multicores from the compiler's perspective: A blessing or a curse?" in *2005 International Symposium on Code Generation and Optimization (CGO)*, 2005. [Online]. Available: http://cag.lcs.mit.edu/commit/papers/05/amarasinghe-CGO-Keynote-Abstract-03-05.pdf

[3] T. Moseley, D. Grunwald, D. A. Connors, R. Ramanujam, V. Tovinkere, and R. Peri, "Loopprof: Dynamic techniques for loop detection and profiling," in *Proceedings of the 2006 Workshop on Binary Instrumentation and Applications (WBIA)*, 2006.

[4] T. Moseley, D. A. Connors, D. Grunwald, and R. Peri, "Identifying potential parallelism via loop-centric profiling," in *Proceedings of the 2007 International Conference on Computing Frontiers*, May 2007.

[5] G. Price and M. Vachharajani, "A case for compressing traces with bdds," in *Computer Architecture Letters; Volume 5, Nov. 2006*, 2006.

[6] J. Giacomoni, M. Vachharajani, and T. Moseley, "FastForward for concurrent threaded pipelines," Univerity of Colorado at Boulder, Tech. Rep. CU-CS-1023-07, 2007.

[7] M. Iyer, C. Ashok, J. Stone, N. Vachharajani, D. A. Connors, and M. Vachharajani, "Finding parallelism for future epic machines," in *Proceedings of the 4th Workshop on Explicitly Parallel Instruction Computing*, March 2005.

[8] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transaction on Computers*, vol. C-35, no. 8, pp. 677–691, August 1986.

[9] J. Giacomoni, J. K. Bennett, A. Carzaniga, M. Vachharajani, and A. L. Wolf, "FShm: High-rate frame manipulation in kernel and user-space," Univerity of Colorado at Boulder, Tech. Rep. CU-CS-1015-07, 2006.

[10] J. E. Smith, "Decoupled access/execute computer architectures," *ACM Trans. Comput. Syst.*, vol. 2, no. 4, pp. 289–308, 1984.

[11] G. Ottoni, R. Rangan, A. Stoler, and D. I. August, "Automatic thread extraction with decoupled software pipelining," in *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*. Los Alamitos, CA, USA: IEEE Computer Society, 2005, pp. 105–118.

[12] Advanced Micro Devices, "AMD completes ATI acquisition and creates processing powerhouse," http://www.amd.com/us-en/Corporate/VirtualPressRoom/0,,51_104_543~113741,00.html, October 2006.