ToolBlocks: An Infrastructure for the Construction of Memory Hierarchy Analysis Tools.

Timothy Sherwood and Brad Calder

University of California, San Diego
{sherwood,calder}@cs.ucsd.edu

Abstract. In an era dominated by the rapid development of faster and cheaper processors it is difficult both for the application developer and system architect to make intelligent decisions about application interaction with system memory hierarchy. We present ToolBlocks, an object oriented system for the rapid development of memory hierarchy models and analysis. With this system, a user may build cache and prefetching models and insert complex analysis and visualization tools to monitor their performance.

1 Introduction

The last ten years have seen incredible advances in all aspects of computer design and use. As the hardware changes rapidly, so must the tools used to optimize applications for it. It is not uncommon to change cache sizes and even move caches from off chip to on chip within the same chip set. Nor is it uncommon for comparable processors or DSPs to have wildly different prefetching and local memory structures. Given an application it can be a daunting task to analyze the existing DSPs and to choose one that will best fit your price/performance goals. In addition, many researchers believe that future processors will be highly configurable by the users, further blurring the line between application tuning and hardware issues. All of these issues make it increasingly difficult to build an suite of tools to handle the problem.

To address this problem we have developed *ToolBlocks*, an object oriented system for the rapid development of memory hierarchy models and tools. With this system a user may easily and quickly modify simulated memory hierarchy layout, link in preexisting or custom analysis and visualization code, and analyze real programs all within a span of hours rather than weeks. From our experience we found that there are three design rules necessary for building any successful system for memory hierarchy analysis.

1. Extendibility: Both the models and analysis must be easily extendible to support the ever changing platforms and new analysis techniques.
2. Efficiency: Most operations should be able to be done with a minimal amount of coding (if any) in a small amount of time.

3. Visualization: Visualization is key to understanding complex systems and the memory hierarchy is no different.

Noticeably missing from the list is performance. This has been the primary objective of most other memory hierarchy simulators [1–3], and while we find that reasonable performance is necessary, it should not be sought at the sacrifice of any of the other design rules (most notably extendibility).

ToolBlocks allows for extendibility through it's object oriented interface. New models and analysis can be quickly prototyped, inserted into the hierarchy, tested and used. Efficiency is achieved through a set of already implemented models, analysis and control blocks that can be configured rapidly by a user to gather a wide range of information. To support visualization the system hooks directly into a custom X-windows visualization program which can be used to analyze data post-mortem or dynamically over the execution of an interactive program.

2 System Overview

ToolBlocks can be driven from either a trace, binary modification tool, or simulator. It is intended to be an add on to, not a replacement for, lower level tools such as ATOM [4] and DynInst [5]. It was written to make memory hierarchy research and cross platform application tuning more fruitful and to reduce redundant effort. Figure 1 shows how tool blocks fits into the overall scheme of analysis.

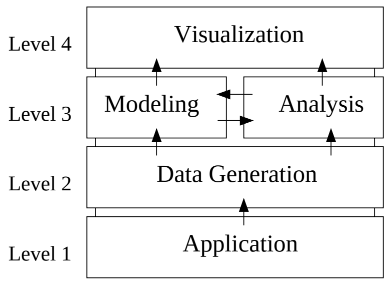

Fig. 1. Typical flow of data in an analysis tool. Data is gathered from the application which is used to do simulations and analysis whose results are then visualized

At the bottom level we see the application itself. It is here that all analysis must start. Level 2 is where data is gathered either by a tracing tool, binary modification tool or simulation. Level 3 is where the system is modeled, statistics are gathered, and analysis is done. At the top level data is visualized by the end user. ToolBlocks does the modeling and analysis of level 3, and provides some visualization.

The ToolBlocks system is completely object oriented. The classes, or *blocks*, link together through *reference streams*. From this, a set of blocks called a *block* *stack* is formed. The block stack is the final tool and consists of both the memory hierarchy simulator and the analysis. At the top of the block stack is a terminator, and at the bottom (the leafs) are the inputs. The stack takes input at the bottom and sends it through the chain of blocks until it reaches a terminator. Figure 2 is a simple example of a block stack.

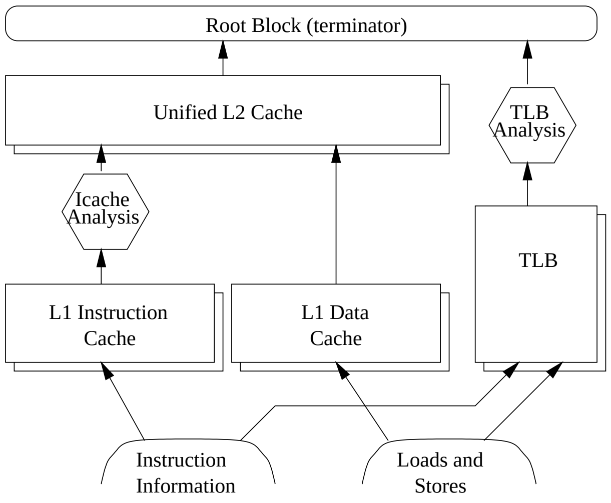

Fig. 2. The basic memory hierarchy block stack used in this paper. Note the terminating `RootBlock`. The inputs at the bottom may be generated by trace or binary modification. The hexagons are analysis, such as source code tracking or conflict detection.

The class hierarchy is intentionally quite simple to support ease of extendibility. There is a base class, called `BaseBlock`, from which all blocks inherit. The base block contains no information, it simply defines the most rudimentary interface. From this there are three major types of blocks defined: model blocks, control blocks, and analysis blocks.

Model blocks represent the hardware structures of the simulated architecture, such as cache structures and prefetching engines. These blocks, when assembled correctly, form the base hardware model to be simulated.

Control blocks modify streams in simple ways to aid the construction of useful block stacks. The simplest of the control blocks is the root block, which terminates the stack by handling all inputs but creating no outputs. However there are other blocks which can be used to provide user configurability without having to code anything. For example filter, split and switch blocks.

The analysis blocks are the most interesting part of the ToolBlocks system. The analysis blocks are inserted into streams but have no effect on them, they simply pass data along, up to the next level without any modifications. The analysis blocks are used to look at the characteristics of each stream so that a better understanding of the traffic at that level can be gained. There are currently four analysis routines, `TraceBlock` for generating traces, `PerPcBlock` for tracking memory behavior back to source code, `HistogramBlock` for dividing up the data into buckets, and `ViewBlock` for generating a visual representation of the data. These analysis routines could further be linked into other available visualization tools.

The total slowdown of program execution varies depending on the block stack, but is typically between 15x and 50x for a reasonable cache hierarchy and a modest amount of analysis and all the ATOM code inserted into the original binary.

2.1 Example Output

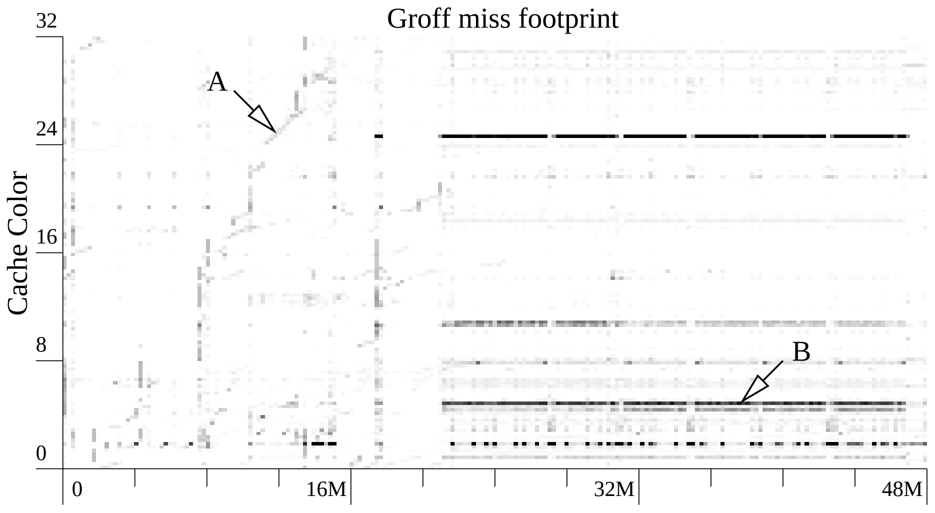

Fig. 3. Original footprint for the application Groff. The x axis is in instructions executed (millions) and the y axis is the division by cache color. Note the streaming behavior seen at point A, and the conflict behavior at point B.

Having now seen an overview of how the system is constructed, we now present an example tool and show how it was used to conduct memory hierarchy research. The tool we present is a simple use of the cache model with a `ViewBlock` added to allow analysis of L2 cache misses. The memory hierarchy is a split virtually indexed L1, and a virtually indexed unified L2. On top of the L2 is a visualization block allowing all cache misses going to main memory to be seen.

Figure 3 shows the memory footprint of the C++ program `groff` taken for a 256K L2 cache. On the X axis is the number of instructions executed, and on the Y axis is a slice of the data cache. Each horizontal row in the picture is a cache line. The darker it is the more cache misses per instruction. As can be seen, there are two major types of misses prevalent in this application, streaming misses (at point A) and conflict misses (at point B).

The streaming capacity/compulsory misses, as pointed to by arrow A, are easily seen as angled or vertical lines because as memory is walked through, sequential cache lines are touched in order. Conflict misses on the other hand are characterized as long horizontal lines. As two or more sections of memory fight for the same cache sets, they keep kicking each other out, which results in cache sets that almost always miss.

From this data, and through the use of the PerPC block, these misses can be tracked back to the source code that causes them in a matter of minutes. The user could then change the source code or the page mapping to avoid this.

3 Conclusion

In this paper we present ToolBlocks as an infrastructure for building memory hierarchy analysis tools for application tuning, architecture research, and reconfigurable computing. Memory hierarchy tools must provide ease of extension to support a rapidly changing development environment and we describe how an powerful and extendible memory hierarchy tool can be built from the primitives of models, analysis, and control blocks.

We find that by tightly coupling the analysis and modeling, and by the promotion of analysis blocks to first class membership, a very simple interface can provide a large set of useful functions. The ToolBlocks system is a direct result of work in both conventional and reconfigurable memory hierarchy research and is currently being used tested by the Compiler and Architecture and MORPH/AMRM groups at UC San Diego. You can retrieve a version of ToolBlocks from `http://www-cse.ucsd.edu/ groups/ pacl/ tools/ toolblocks.html`. This research was supported by DARPA/ITO under contract number DABT63-98-C-0045.

References

1. Sugumar, R., Abraham, S.: Cheeta Cache Simulator, From University of Michigan.
2. Hill, M., Smith, A.: Evaluating Associativity in CPU Caches. IEEE Trans. on Computers, C-38, **12**, December 1989, p.1612–1630.
3. Gee, J., Hill, M., Pnevmatikatos, D., Smith, A. Cache Performance of the SPEC Benchmark Suite. IEEE Micro, August 1993, **3**, 2.
4. Srivastava, A., Eustace, A.: ATOM: A System for Building Customized Program Analysis Tools. Proceedings of the Conference on Programming Language Design and Implementation, pages 196-205. ACM, 1994.
5. Hollingsworth, J., Miller, B., Cargille, J.: Dynamic Program Instrumentation for Scalable Performance Tools In the Proceedings of 1994 Scalable High Performance Computing Conference, May 1994.