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. 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. We propose a framework for customizing the cache configurations to individual programs using *dynamic way allocation* (DWA). We then describe cache configuration methodologies that permit DWA.

2 The Proposed Reconfigurable Cache Model

DWA 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. DWA 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. 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, maximizing 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. We present 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 pro-grams have no idea of available cache size at compile time, and neither target runtime environment nor program behavior can be determined at compile time. The software configuration routine provides an alternative till such a compiler is available.

The software routine overcomes the drawbacks of static reconfiguration. It also allows reconfiguration of caches for legacy code or code that cannot be recompiled. To be effective, the software routine may need to run frequently, and 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 [2]. Our model uses common runtime statistics such as IPC, miss rate, and cache references, that performance counters already provide, plus additional parameters like operating temperature and power dissipation statistics to trigger reconfiguration. These are held in dedicated registers, and are accessed by the software routine to determine when and how to reconfigure the cache.

For e.g., 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

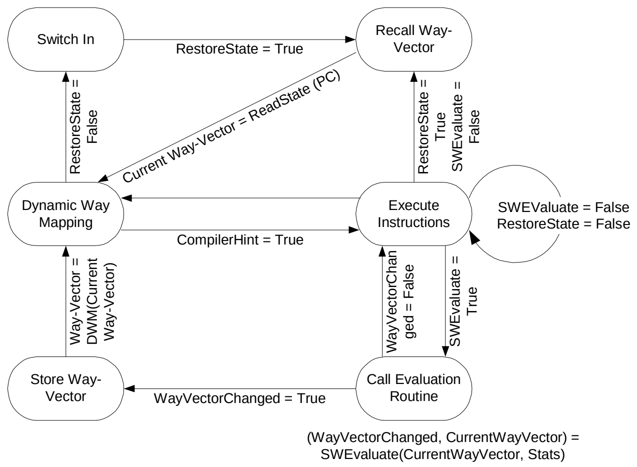

Fig.1: Reconfiguration operations of the proposed cache architecture

operating temperature). We could then write 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 [3]. 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* [2], [4]. 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}. 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 [5] 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", *CSG Memo 430*, MIT

[2] D. H. Albonesi, "Selective Cache Ways: On-Demand Cache Resource Allocation", *Journal of ILP*, May 2000.

[3] Michael Powell *et al.*, "Gated-V_{dd}: A Circuit Technique to reduce leakage in Deep-Submicron Cache Memories", In *Proc. of the ISLPED*, July 2000.

[5] P. Ranganathan, S. Adve, N. Jouppi, "Reconfigurable Caches and their Application to Media Processing", *ISCA 2000*.

[6] D. Brooks, *et al.*, "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations", *ISCA 2000*