Performance Enhancement by Eliminating Redundant Function Execution

Peng Chen, Krishna Kavi and Robert Akl
Computer Science and Engineering Department
University of North Texas
{pc0043, kavi, rakl}@cs.unt.edu

Abstract

Programs often call the same function with the same arguments, yielding the same results. We call this phenomenon, "function reuse". Previously, we have shown such a behavior for some of the SPEC2000 integer benchmarks using HP ATOM instrumentation tools. However, this required extensive analysis by hand, and assumptions regarding side-effects caused by functions. In this paper, we modified a well-known architecture simulator, SimpleScalar, to analyze multiple benchmarks to investigate the function reuse behavior.

Key words. *Function reuse, Speculative Execution, Value Prediction, Instruction Reuse, Basic Block Reuse, Instruction Level Parallelism, SimpleScalar*

1. Introduction

In modern computer systems, CPUs are endowed with multiple processing elements (such as multiple floating point and integer arithmetic units, memory access units, branch units). The goal then becomes the execution of multiple instructions per cycle, as characterized by the desire to increase Instructions Per Cycle (IPC) executed by processing elements. The basic idea is to execute "independent" instructions from a program in parallel, potentially in an order other than that defined by the program sequencing. It was soon discovered that insufficient instruction level parallelism (ILP) existed in a single thread, leading to the utilization of multiple threads (either from the same program or from a workload consisting of different applications running on the CPU). Modern processors also rely on concepts such as speculative execution of instructions based on branch prediction techniques. Such techniques do increase IPC, but do not necessarily increase the number of useful instructions executed.

Related to branch prediction, that aims to predict the path a program sequence takes, is value prediction. Value prediction [1] originally predicted that "load" instructions that read data from a memory location, return the same value as a previous memory access to the same address. Since memory accesses take much longer than other instructions [2], reusing a previously fetched value (based on the prediction that the values are the same) can permit the CPU to achieve higher IPC counts. Techniques such as "trace caches" [3] can help with value prediction, since trace caches retain histories of execution of instructions. As a next step in this trend, one can use prediction for memory addresses [4] since the addresses of array elements often differ by a constant displacement; and even predicting that an arithmetic instruction (such as an ADD) produces the same result as a prior execution of the instruction, particularly involving loops where the same instruction is executed repeatedly, with the same register operands, (provided that the registers are not modified). While compiler optimizations move loop invariant code out of a loop, some invariant code cannot be detected at compile time. More recently, this phenomenon of predicting the results of an instruction execution is extended to a sequence of instructions (i.e., basic block). A basic block is a sequence of instructions with one entry point at the beginning of the block and one exit at the end of the block. If the basic block of instructions accesses a limited number of register operands, it is possible to assume that the repeated executions of the block with the same register operands [5] (assuming the registers are unchanged between the executions) will yield the same results.

It should be noted that the underlying theme in all these techniques is the ability of the architectures to execute instructions (or blocks of instructions) [6] in a speculative mode, and quashing the result when the speculation fails. While all speculative techniques increase IPC counts, it should be noted that the effective execution time of an application is not reduced (and in some cases it is increased to account for the time needed to undue an incorrect speculation).

Our approach is a continuation of these techniques [7] [8], by extending the reuse of results (and prediction of results) from repeated execution of functions [9]. We believe that, at least for integer applications, functions are often invoked repeatedly with the same operands, producing the same results. We propose to eliminate the redundant function executions, improving the performance. It should be noted that our approach eliminates the execution of instructions comprising a function, and thus may result in lower IPC counts, but shorter execution times.

In modern programming language, such as C, C++, or Java, most of the programming codes are constructed in well-structured modules such as functions, methods or procedures. The tendency of programming evolution is to make program code more readable and easy to maintain. In large programs, some functions may be called multiple times with the same input set and generate the same outputs. Our motivation is based on empirical observations suggesting that results of many functions, having the same inputs, can be reused instead of being executed repeatedly.

Previously, we used ATOM [10] to instrument code to detect the function reuse phenomenon. Our results [11] show significant opportunities for function reuse, particularly for integer benchmarks. These studies were aimed at the feasibility and not intended to evaluate performance gains possible from reusing the results of function execution. In this paper we report our studies that involve modifications to a well-known computer architecture simulator called SimpleScalar.

SimpleScalar [12] was written in 1992 as part of the Multiscalar project at the University of Wisconsin. In 1995, with Doug Burger's assistance, the toolset was released as an open source distribution freely available to academic and noncommercial users. SimpleScalar LLC now maintains the suite of tools, and is distributed through SimpleScalar website - http://www.simplescalar.com.

Since its release, SimpleScalar has become popular with researchers and instructors in the computer architecture research community. For example, in 2000 more than one-third of all papers published in top computer architecture conferences used SimpleScalar tools to evaluate their designs. The use of a common simulator infrastructure permits researchers to exchange designs as well as to repeat experiments to validate claims independently.

SimpleScalar provides an infrastructure for computer system modeling that simplifies implementing hardware models capable of simulating complete applications. During simulation, model instrumentation measures the dynamic characteristics of the hardware model and the performance of the software running on it. All SimpleScalar modules use execution-driven simulation techniques. For computer system models, this process requires reproducing the execution of instructions on the simulated machine. A popular alternative, trace-driven simulation employs a stream of prerecorded instructions to drive a hardware-timing model. This method uses a variety of hardware- and software-based techniques—such as hardware monitoring, binary instrumentation, or trace synthesis—to collect instruction traces. This approach is often used for evaluating cache memory designs. Execution-driven simulation provides many advantages compared to trace-based techniques. Foremost, the approach provides access to all data produced and consumed during program execution. These values are crucial to the study of optimizations such as value prediction, compressed memory systems, and dynamic power analysis. In dynamic power analysis, the simulation must monitor the data values sent to all microarchitectural components such as arithmetic logic units and caches to gauge dynamic power requirements. The Hamming distance of consecutive data inputs defines the degree to which input bits change, which in turn causes transistor switching that consumes dynamic power.

SimpleScalar includes several simulation modules [12] suitable for a variety of common architectural analysis tasks. The simulators range from *sim-safe*, a minimal SimpleScalar simulator that emulates only the instruction set, to *sim-out-of-order*, a detailed microarchitectural model with dynamic scheduling, aggressive speculative execution, and a multilevel memory system. All the simulators have fairly small code sizes because they leverage SimpleScalar's infrastructure components, which provide a broad collection of routines to implement many common modeling tasks, including instruction-set simulation IO emulation, discrete-event management, and modeling of common micro architectural components such as branch predictors, instruction queues, and caches. In general, the more detailed a model becomes, the larger its code size and the slower it runs due to increased processing for each instruction simulated.

In order to prove our idea, we set up our experiments running on a modified SimpleScalar. More specifically, we modified Sim-Safe functional simulator business logic and introduced function reuse and prediction tables into the in-order execution pipeline to investigate the potential performance gains from function reuse for SPEC2000 integer benchmarks and MiBench embedded integer benchmarks. In the near future, we are planning to extend our experimentation to out-of-order execution and multithreaded systems.

2. Related Work

The performance of modern computer architectures that include multiple functional units is highly dependent on the instruction level parallelism (ILP) that can be extracted from a sequential program. The ILP is limited by control and data dependencies. Branch prediction tries to help with control dependencies. Numerous branch prediction techniques have been proposed and implemented in modern architectures.

Value prediction in a similar vein attempts to overcome “true” data dependencies. In [13] the authors have studied the performance of different value predictors for speculative multithreaded processors. The paper proposed a value predictor called the increment predictor, and evaluated its performance for a clustered speculative multithreaded architecture. The goal of this technique is to predict the increment values, particularly for address computations. It was claimed that a 1-KB increment predictor achieves 73% prediction accuracy and a performance that is just 13% lower than that of a *perfect* predictor with infinite sized buffers.

Sodani and Sohi [14] save the source and destination register values for each instruction and allow instruction execution to be skipped if the current instruction input values match previously cached values for that instruction. This schemes showed 34% reuse of instructions, and this reuse is directly dependent on the size of the reuse buffer. With sufficiently large buffers, program execution times were reduced by 20%.

Jian Huang and David J. Lilja [15] have focused on basic-block reuse. They investigated the input and output values of basic blocks and found that these values were predictable. For SPEC benchmark programs evaluated, 90 percent of the basic blocks had fewer than four register inputs, five live register outputs, four memory inputs, and two memory outputs. About 16 to 41 percent of all the basic blocks were repeating earlier calculations when the programs are compiled with the -O2 optimization level in the GCC compiler.

In our previous work [11], we set up a look-up table for functions to “cache” the behavior of functions, in order to find out how often repeated invocations of functions involve identical arguments. Function reuse is applicable only for side-effect free functions. By using HP(DEC) ATOM to instrument benchmark programs to evaluate our idea, we showed that the potential function reuse rate for SPEC2000 integer benchmarks could range from 7% to 66%. ATOM permits us to trace the execution of programs with instrumented code. Thus our previous experiments allowed us to evaluate the frequency of function reuse. ATOM does not allow us to measure execution times for programs, since the instrumentation negatively affects the cycle counts. In this paper we show our results using SimpleScalar simulator.

3. Motivation for Function Reuse

Our work is motivated by our observations on recursive computations in the context of our multithreaded architecture known as SDF [16]. In our architecture a new thread is spawned for each function (including recursive functions) and basic blocks. We observed that the dynamically spawned threads sometimes replicate their computations since they are invoked with exactly the same input values as other threads. Our SDF architecture is based on dataflow, and functions are side-effect free, making it easier to implement the concept of function reuse. This is more difficult with imperative systems that use pointers and shared global variables. However compile time analysis can identify functions that are side-effect free. For now, we pre-analyze functions by hand for their freedom from side-effects, and identify candidates for reuse.

Consider a simple recursive Fibonacci function shown below.

```
#include <stdio.h>
int fib (int);
int main()
{   int num = 30;
    printf ("The value is %d .\n ", fib (nun
    return 0;
}
int fib (int num)
{  if (num == 1|| num == 2) return 1;
      else
      return fib (num-1) + fib (num-2);
}
```

In MIPS-like architecture, when a function call is reached, a stack frame is placed on the run-time stack. As can be observed, the example above will create multiple copies of identical stack frames in order to complete a fib(n) since fib(n) and fib(n-1) both spawn fib(n-2). If we can create a look-up table containing the recursive calls by storing the input value of the previous functions and the results, then future calls with identical input values can be skipped in the pipeline. This innovation could not only save significant CPU computation time, but save power consumed by a program as well.

As can be seen from the Table 1, at least for the function at hand, a significant performance can be gained by reusing the values produced by prior invocations of the function with identical input values. The table also shows how many times functions are invoked with identical values. We used modified SimpleScalar Sim-Safe simulator to obtain these results.

4. Experiment Results

Our modified SimpleScalar was installed on PC/Linux P4 2GHz machines. The reason for picking Sim-Safe simulator instead of Sim-Out-of-Order is due to its simpler pipeline structure with all necessary functional behaviors. We are planning to extend our experimentation to out-of-order execution and multithreaded systems in the near future.

4.1. Benchmarks

Benchmark programs analyzed in this paper are listed in Table 2 along with their inputs. There are four integer benchmark programs (*dijkstra, rawcaudio, bit counts, quick sort*) from MiBench, two from SPEC 2000 (*parser, 176.gcc*) and five integer programs from SPEC '95 suite (*go, m88ksim, vortex, ijpeg and perl)*. We also included Fibonacci in our discussions. These programs run either to completion or up to 4 billion instructions. All the programs were compiled using GNU gcc (version 2.6.3).

Table 1. Function reuse for Fibonacci function

Function	Clock Cycles (Without reuse)	Clock Cycles (With reuse)	# times the Function is Called	# times the Function is Reused with same Inputs
Fib (10)	31,124	10,200	895	609
Fib (20)	3,034,273	1,137,378	980,560	637,364
Fib (30)	290,056,432	98,456,320	920,876,456	616,987,225

Table 2. Benchmark Programs Used

Benchmark Name	Benchmark Names	Input
	Fibbonnacci	in
MiBench	Dijkstra	Large
	Rawcaudio	Large
	Bit Count	Large
	Quick sort	Large
SPEC '2000	Parser	Ref.in (training)
	Gcc	200.i (reference)
SPEC '95	Perl	Scrabbl.in (training)
	Ijpeg	Vigo.ppm (training)
	Vortex	Vortex.in (training)
	M88kSim	Ctl.in (reference)
	Go	Null.in (reference)

4.2 Experiments and Results

We performed several experiments to evaluate the concept of dynamic function reuse. Here we only concentrate on some key initial results for some sample configurations of the proposed mechanisms. We conducted experiments to discover how often functions are invoked with the same input arguments

Figure 1 shows how often functions are invoked with the same inputs as prior invocations. The reuse buffer was designed with unlimited capacity to capture all the repeated function invocations. The data shows that for the benchmarks examined, between 7.29% to 67.7% function invocations can be skipped by using buffered function results. This indicates that there is a great potential for function reuse in improving execution performance of not only larger benchmarks such as SPEC programs but also of small embedded integer such as MiBench. In fact, Figure 1 shows that small integer benchmarks always generate high and stable function reuse rates. Since CPU computation time is a key metric of interest to computer systems designers, the next experiment we conducted on SimpleScalar is to show how much CPU time could be actually saved compared to normal executions without function reuse. At first, we assume the function reuse look-up time takes one clock cycle. This presents an upper limit on achievable performance gains. The speedup shown indicates the comparison between benchmark execution with function reuse and without function reuse.

Function-reuse buffer is usually a device similar to a data cache. The speed of its decision-making depends on its internal logic design. In order to see how the buffer access speed impacts the system performance (CPU time saved), we repeated the experiment by setting the look-up time (to determine if a function execution can be skipped) to 10 clock cycles. Most functions usually contain a large number of instructions, consuming more 10 cycles. Comparing the data in Table 3 with the data in Table 4, we note that the speedup decrease due to increased buffer look-up time is only slight.

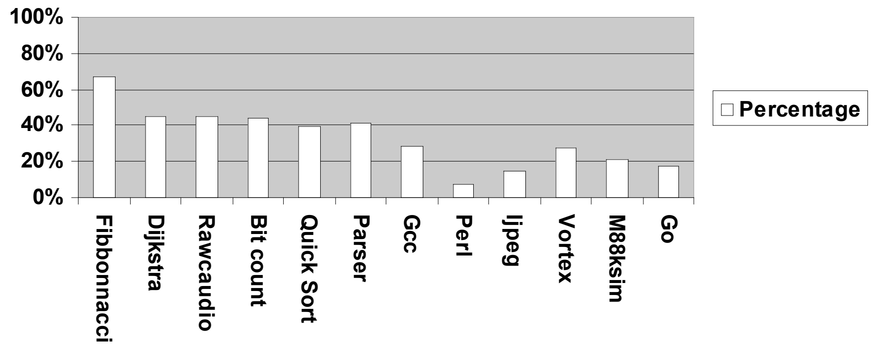

Figure 1. Function Reuse Rate

Table 3. Execution Speedup Achieved with one cycle

Benchmark	Speedup
Fib	3.23
Dijkstra	1.83
Rawcaudio	1.81
Bit Count	1.81
Quick Sort	1.67
Parser	1.71
Gcc	1.40
Perl	1.22
Ijpeg	1.27
Vortex	1.42
M88ksim	1.38
Go	1.37

Table 4. Execution Speedup Achieved with ten cycles

Benchmark	Speedup
Fib	3.10
Dijkstra	1.76
Rawcaudio	1.72
Bit Count	1.74
Quick Sort	1.55
Parser	1.62
Gcc	1.26
Perl	1.20
Ijpeg	1.23
Vortex	1.32
M88ksim	1.33
Go	1.29

In the previous experiments, the reuse buffer size was assumed unlimited in size. In real microprocessor environment we need to rely on finite sized tables or buffers to cache functions. In order to simulate the impact of finite sized buffers, we treat the buffer as a cache and explore different sizes and associativities for the buffer. We used 128, 256, 512 and 1024 entry reuse buffers (with sufficient width to cache all input arguments for functions and results). In Figure 2, we show the speedup of using direct-mapped cache as the buffer. In Figure 3 we show results using 2-way set-associative reuse buffers. Figure 4 shows the data for fully associative reuse buffers. Note that the data in Figures 2-4 reflect the speedup of function reuse.

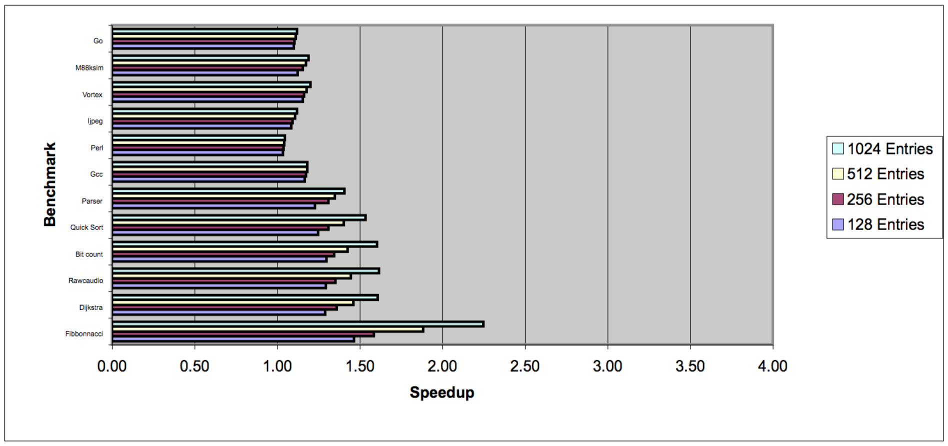

Figure 2. Function reuses for direct-mapped reuse buffer

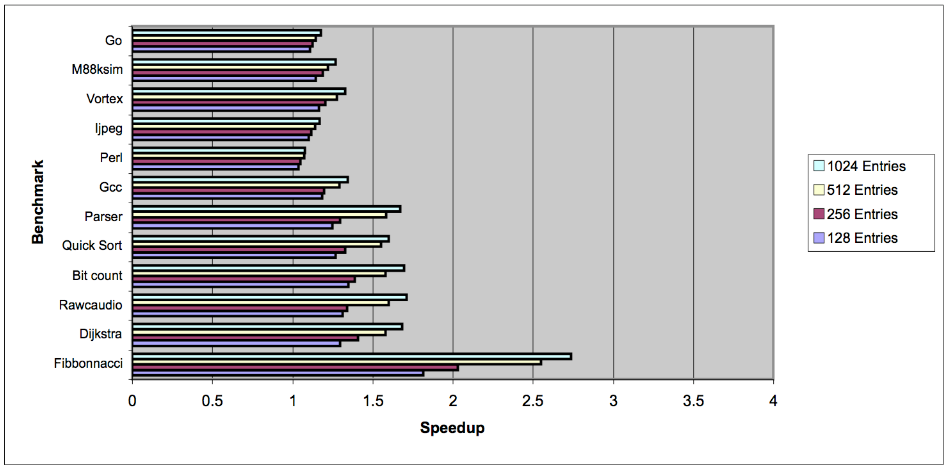

Figure 3. Function reuses for 2-way set associative reuse buffer

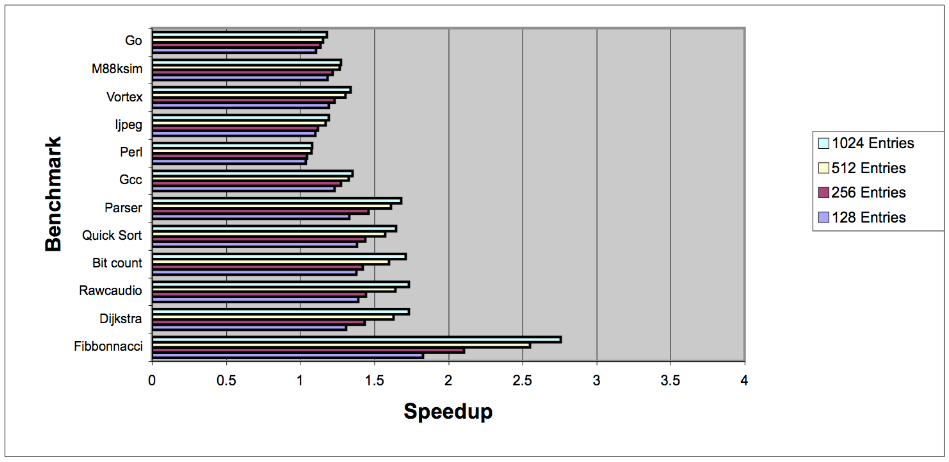

Figure 4. Function reuses for full-way set associative reuse buffer

It is obvious that as the buffer size increases, we can capture more function invocations that can be reused. When higher set associtivities are used, conflicts in storing information about invoked functions are minimized. This is reflected in higher speedup of reuse in Figures 3 and 4 when compared to that in Figure 2. Interestingly, Figure 4 does not show significant improvement in function reuse speedup since larger buffer size (such as 1024 entries) of 2-way set associative reuse buffer are adequate for our purpose. This behavior is similar to that observed with set associative data caches.

5. Micro-architecture and scheme with reuse buffer

In this section we outline how the concept of function reuse can be incorporated into modern architecture pipelines. Function reuse is a non-speculative technique that exploits (dynamic) redundancy in programs by obtaining results of functions based on their prior executions, and thereby skipping repeated executions. The organization of the reuse buffer is shown in Figure 5. Figure 6 shows a pipeline with function reuse. When a function is first executed, the signature of the function and its results are stored in a hardware structure called a Reuse Buffer (Figure 5). The buffer is indexed by the program counter of the function call (such as JAL or JSR). The reuse buffer is accessed concurrently with instruction fetch. If an entry is found (indicating that the instruction is a function call), the entry must be checked for matching input values. Function arguments are defined in registers based on programming convention used by the compiler. For example on most MIPS based systems; arguments are available in registers R4-R7. When a function with the same inputs is encountered again, its previous results are read from the buffer and they are directly saved in the registers designated for function results (e.g., R2 and R3 in MIPS convention). When this occurs, the function call is squashed and the program is continued from the instruction beyond the return (by simply incrementing the program counter). When function is not reused, a new entry for the function with new arguments is cached in the buffer, and results of the function execution (when return instruction is committed) are added to the buffer for possible future reuse. The function results are available in specified registers (R2 and R3 in MIPS).

PC	Input 1	Input 2	Input 3	Input 4	Return value

Figure 5. Reuse Buffer Entry

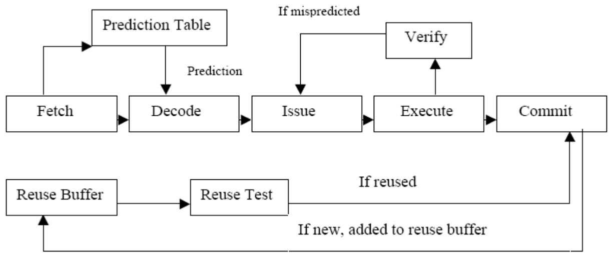

Figure 6. Function reuse architecture

6. Conclusions

In this paper we verified the concept of dynamic function reuse on modified SimpleScalar simulator with our hardware device, function reuse buffer incorporated in it. This research continues recent trends in architecture that have investigated speculation, branch prediction, value and address prediction, instruction reuse and basic block reuse. Function reuse is applicable for side-effect free (or pure) functions. Our results show that the function reuse idea is not only applicable in complex integer benchmark such as SPEC 2000 but works for small integer benchmark MIBench as well. Our studies with integer benchmarks indicate the following:

1. There is a great potential for exploiting function reuse.
2. The performance gains depend on the size and organization of the reuse buffer.
3. Function reuse look-up time does not have a significant impact on system performance.

We believe that object-oriented programming promotes code reuse, which is the reason for function reuse. We proposed a high level architecture for implementing the function reuse concept. In order to fully exploit the idea we need to calculate the power consumption savings since power is a major issue with embedded systems. We will further extend the idea to include function prediction (instead of reuse) in a multithreaded environment. We also plan to extend our function reuse even further to thread level reuse in the future.

7. References

[1] A. Moshovos and G. Sohi; "Read-After-Read Memory Dependence Prediction"; *32nd International Symposium on Micro architecture (MICRO 32)*, pages 313-326, Nov 1999

[2] B. Calder, G. Reinman, and D. Tullsen; "Selective Value Prediction"; *26th International Symposium on Computer Architecture*, pages 64-74, May1999

[3] F. Gabbay and A. Mendelson; "Can Program Profiling Support Value Prediction"; *International Symposium on Microarchitecture (MICRO 97)*, December 01-03, pages 270-280,1997

[4] J. Huang, Y. Choi, and D. Lilja; "Improving Value Prediction by Exploiting Operand and Output Value Locality"; *University of Minnesota Technical Report: HPCA-99-06*, pages 106-114, June 1999

[5] J. Huang and D. Lilja; "Improving Instruction-Level Parallelism by Exploiting Global Value Locality"; *University of Minnesota Technical Report: HPPC-98-12*, Oct 1998

[6] Martin Burtscher and Benjamin G. Zorn; "Prediction Outcome History-basedConfidence Estimation for Load Value Prediction"; *Journal of Instruction-Level Parallelism (JILP)*, Vol. 1, May 1999

[7] Stephen E. Richardson; "Caching Function Results: Faster Arithmetic by Avoiding Unnecessary Computation"; *SMLI TR-92-1*, September 1992

[8] Wall D. W., "Limits on instruction-level parallelism," *Proc. of 4th Intl. Conf. on Architecture Support for programming Languages and Operating Systems (ASPLOS-4),* April 1991, pp.176-188.

[9] A. Sodani and G. Sohi; "Dynamic Instruction Reuse"; *24th International Symposium on Computer Architecture (ISCA),* pages 194-205, June1997

[10] ATOM User Manual PRELIMINARY DRAFT, June, 1995 *Digital Equipment Corporation* Maynard, Massachusetts

[11] K.M. Kavi and P. Chen. "Dynamic function result reuse", *Proceedings of the 11th International Conference on Advanced Computing (ADCOM-2003),* Coimbatore, India, Dec. 17-20, 2003.

[12] D. Burger, T. Austin, and S. Bennett, "The SimpleScalar Tool Set, Version 2.0," *Technical Report 1342*, Computer Science Dept., Univ. of Wisconsin, Madison, June 1997.

[13] Pedro Marcuello, Jordi Tubella, Antonio Gonzalez, "Value Prediction for Speculative Multithreaded Architectures", page 230, *32nd Annual International Symposium on Microarchitecture*, 1999.

[14] A. Sodani and G. Sohi; "An Empirical Analysis of Instruction Repetition"; *8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII*), Oct 1998

[15] J. Huang and D. Lilja; "Exploiting Basic Block Value Locality with Block Reuse"; *University of Minnesota Technical Report: HPPC-98-09*, 1998

[16] K.M. Kavi, R. Giorgi and J. Arul. "Scheduled Dataflow: Execution paradigm, architecture and performance evaluation", *IEEE Transactions on Computer*, Vol. 50, No. 8, pp 834-846, Aug. 2001.