Dynamic Function Result Reuse

Krishna Kavi and Peng Chen
Computer Science and Engineering Department
University of North Texas
Denton, Texas 76203
{kavi, pc0043}@cs.unt.edu

Abstract

Continuing recent architectural trends that rely on branch predictions, value predictions, speculative execution, and reuse of results from instruction execution, we investigate the reuse of results from previous function invocations. In this paper we show that for integer benchmarks, it is possible to eliminate some function invocations since the same function is executed repeatedly with the same arguments. We feel that along with compiler techniques such as function cloning and partial evaluations, dynamic, hardware based techniques to check if a function should be invoked or the results from a prior execution can be used will lead to dramatic performance gains. We use HP Alpha based instrumentation tool called ATOM in our experiments on SPEC2000 integer benchmarks.

Key words. Speculative Execution, Value Prediction, Instruction Reuse, Basic Block Reuse, Instruction Level Parallelism

1. Introduction

The performance of modern computer systems is measured in terms of the number of instructions per cycle (IPC). Modern architectures rely on instruction-level parallelism to improve performance by executing multiple instructions every cycle, often executing instructions in an order other than that specified by the program. Alternatively, multiple independent instructions are packed into a wide instruction word (VLIW) so that the component instructions can be executed simultaneously. In order to increase the number of instructions that can be issued (either statically in VLIW or dynamically in superscalar systems), speculative execution is often employed. Numerous techniques exist to predict conditional instructions in an attempt to insert instructions into the execution pipeline as early as possible and minimize the pipeline stalls. Similar to branch prediction, value prediction and address predictions have received considerable interest in recent years. Continuing this trend, most recently, researchers are exploring "instruction reuse" and "basic block reuse" to eliminate the need for execution of instructions if they produce results that are identical to some prior execution. In this paper we want to extend the scope of reuse to function level.

Programming has gone through several paradigms, from machine code to assembly to procedural programming to object-oriented programming. The tendency of the evolution of programming languages is to make programs more readable and easy to maintain. The programs are constructed using modules and functions emphasizing code reuse. In a large and complex program, the same function may be called multiple times. Sometimes, those functions will take the same set of inputs and produce the same output. Empirical observations suggest that results of many functions, having the same inputs, could be reused instead of being executed repeatedly.

Compiler techniques exist to statically eliminate redundant computations, or replace dynamic computations with static computations. Function cloning and partial evaluations are examples of such techniques. We believe that hardware techniques can be used to "cache" the behavior of functions and further eliminate repeated executions of functions with same arguments. Function reuse is applicable only to "pure" or side-effect free functions. In this research we are investigating the potential performance gains from function reuse for SPEC2000 integer benchmarks. We use ATOM to instrument benchmark programs to evaluate our idea.

2. Related Research

The performance of modern computer architectures that include multiple functional units is highly dependent on the instruction level parallelism that can be extracted from a sequential program. The ILP is limited by control and data dependencies. The architecture research has proposed numerous branch-prediction techniques to overcome the control dependencies (see [H&P] for detailed descriptions of several of these techniques and their impact on ILP). Value prediction [Gabby-Mendelson, Lipsti, Smith], in a similar vein attempts to overcome "true" data dependencies. In this case, a dependent instruction is provided with a predicted value without having to wait for the actual execution of a predecessor instruction. The value prediction is particularly effective on Read-after-Read memory dependencies [moshovas-sohi]. Such dependencies refer to successive load instructions that access the same memory location without any intervening stores to the location. While compilers can eliminate some loads, the window over which a compiler can explore its optimization is often limited a basic block or a function. Hardware based value prediction can further expose and eliminate redundant loads.

Martin and Benjamin [11] show that Load instructions occasionally incur very long latencies that can significantly affect system performance. Load value prediction alleviates this problem by allowing the CPU to speculatively continue processing without having to wait for the slow memory access to complete. Current load value predictors can only correctly predict about forty to seventy percent of the fetched load values. To avoid the cycle-penalty for mispredictions in the remaining cases, confidence estimators are employed. They inhibit predictions that are not likely to be correct. The authors of the paper [11] also present a novel confidence estimator that is based on prediction outcome histories. Profiles are used to identify high-confidence history patterns. The confidence estimator is able to trade-off coverage for accuracy with great flexibility and reaches average prediction accuracy of 99.3% for some SPECint95 benchmarks. Cycle-accurate pipeline-level simulations show that a simple last value predictor combined with confidence estimator outperforms other predictors.

Value reuse and instruction reuse are the next evolutions where the results of instruction executions are saved in reuse buffers or result caches and the results are reused in future computations [Sodani-Sohi, Richardson, UMN reports]. The difference between value prediction and value (or instruction) reuse should be understood. On a prediction, the computation still needs to be executed and the predicted value must be verified (and re-execute all computations based on mis-predicted values), while on a reuse, the computation is not executed. Richardson [12] uses a cache called result cache to save the results of floating point operations**.** The index of the cache is obtained by hashing the operand values. The result cache is accessed in parallel with the execution of a floating-point operation. If the result is found in the result cache then the operation is squashed.

For this purpose of reusing instructions dynamically Sodani and Sohi [3] propose to save the source and destination register values for each instruction and allow instruction execution to be skipped if the current instruction input values match a previously cached values for that instruction. The reuse buffer schemes they created 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%.

Some other researchers have focused reuse at basic-block level [9]. It should be noted that unlike a single instruction, a basic block consists of a large number of inputs (the values of live registers at the time the basic block is entered) and outputs (values of live registers on block exit). In order to truly exploit basic block reuse, one needs to buffer all input and output values. In order to minimize the hardware complexity, the concept of basic block locality is introduced. A depth-1-inputs (or outputs) indicates that the immediately previous execution of the basic block used the same inputs (or outputs). Likewise, depth-n-inputs indicate the nth previous execution of the basic block was with identical inputs. Depth-1-inputs showed between 2% to 41% localities while depth-1-outputs showed between 3% and 51% localities. Higher depths (with larger buffers to cache several previous inputs and outputs of the same block) resulted in higher localities. Simulation results showed that a 2048-entry *block history buffer* with enough input and output fields to cover the requirements of 90% of the basic blocks produced miss rates below 7%. Block reuse with this *block history buffer* improved performance for the tested programs from 1% to 14% with an overall average improvement of 9% when using reasonable hardware assumptions.

In this paper we extend these previous works to reuse at function level. We limit ourselves to integer benchmarks. We feel that function reuse (as well as block reuse) is less effective with floating point programs. Function value prediction may be utilized for such cases whereby one thread can continue execution beyond the function utilizing a predicted value while a separate thread is used to execute the function and squash computations based on mis-predicted values.

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 [14]. In our architecture a new thread is spawned for each function (including recursive functions). We observed that the dynamically spawned threads sometimes replicate their computations since they are invoked with exactly the same input values as other threads. Consider a simple recursive Fibonnacci function shown below.

```
#include <stdio.h>
int fib(int);
int main()
{
    int num = 30;
    printf("The value is %d .\n ", fib(num) );
    return 0;
}

int fib(int num)
{
    if(num == 1) return 1;
    if(num == 2) return 1;
    else
        return fib(num-1) + fib(num-2);
}
```

As can be observed, this implementation of the function creates multiple copies of fib(1), fib(2)…, since fib(n) and fib(n-1) both spawn fib(n-2). Of course it is possible to rewrite the Fibbonnacci function to use iteration instead of recursion. However, one can also consider the use of a buffer that caches the recursive calls by storing the input value of the function and the result so that future calls with identical input values can be skipped. Table 1 shows the results of reusing functions for Fibbonnaci functions. We used ATOM instrumentation tool to obtain these results. As can be seen from the table, 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 (last column).

Table 1: Function Reuse for Fibbonnacci

Function	Clock cycles (without reuse)	Clock cycles (with reuse)	Function called	Function called more than 1 with same inputs
Fib (10)	106,742	100,655	1,090	730
Fib(20)	3,611,713	263,757	135,290	87,938
Fib(30)	454,250,016	33,173,110	132,9000	890,430
Fib(40)	53,933,197,903 (54 billion)	3.94 billion	127,278,861	86,167,788

Given these results, we wanted to explore the concept of function reuse to a wider range of applications.

4. Experimental evaluation

Our experiments are conducted on DEC/HP Alpha machines, which provide a very useful and powerful instrumentation tool, ATOM [15]. In order to test the behaviors for different organizations of the reuse buffer, we used, Dinero cache simulator. Commercial applications are frequently large and poorly understood programs. Performance depends on many factors including program organization, algorithm, compiler optimizations, run-time system, operating systems, instruction set, hardware architecture, cache memories and so on. The key to improving performance is to understand dynamic behavior of these programs. Unfortunately, gathering statistics about program behavior is often difficult without special purpose tools and hardware probes. ATOM provides a flexible code instrumentation interface that is capable of building a wide variety of interesting program analysis tools.

4.1 Benchmarks

The benchmark programs analyzed in this paper are listed in Table 2 along with their inputs and number of dynamic instructions executed. There are two integer benchmark programs (197.parser, 176.gcc) from SPEC '2000 and five integer programs from SPEC '95 suite (*go, m88ksim, vortex, ijpeg and perl)*. We also include Fibonnacci in our discussions. These programs were run either to completion or up to 4 billion instructions. All the programs were compiled using GNU gcc (version 2.6.3) with optimization (-O3).

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

Table 2. Benchmark Programs Used

Benchmarks	Input	Inst. Count (Mil.)
fib	in	643.2
parser	Ref.in(training)	4,000
gcc	200.i(reference)	4,000
Perl	Scrabbl.in(training)	555.6
Ijpeg	Vigo.ppm(training)	442.2
Vortex	Vortex.in(training)	900.0
M88ksim	Ctl.in(reference)	1050.0
go	Null.in(reference)	750.0

with identical input values, how many different (or unique) input combinations are used for most heavily invoked functions, how the finiteness of reuse buffer impacts the performance gains, how the number of function inputs effects the reuse and the impact set-associativity of the reuse buffer. It should be emphasized that function reuse is effective for "pure" or side-effect free functions. In a realistic implementation, compilers can provide information on the nature of the function so that the hardware can cache only those functions that provide high reuse. In our experiments we manually examined functions for side-effects and explored function reuse only for side-effect free functions. Since the benchmarks used are designed with a large number of functions, it become very unwieldy to capture all functions in the reuse buffer. We first analyzed all functions and conducted further investigations only for the 20 most frequently called functions for each benchmark.

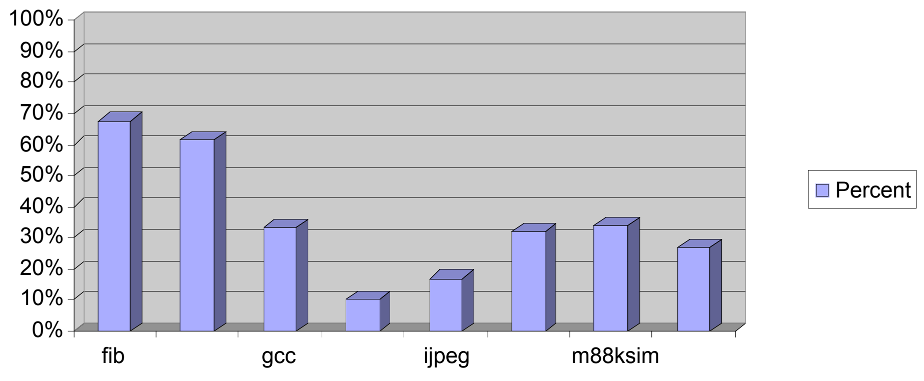


Figure 1. Function Reuse

Figure 1 shows how often the 20 most frequently called functions are invoked with the same inputs as prior invocations. The data shows that for the benchmarks examined, from 10% to 67.7% function invocations can be skipped by buffering function results. This indicates that there is a great potential for function reuse in improving execution performance of even large benchmarks.

In the next experiment we wanted to explore when a function reuse occurs. We first thought that higher reuse occurs during initialization. We divided the execution of a program into four quarters and explored the reuse of functions in each of these quarters. If the reuse behavior is different in different segments of the program execution, we may consider different policies for buffer management during the different program execution phases. Figure 2 shows the results. The results are very surprising and indicate that the reuse behavior is rather uniformly distributed over all phases of a program execution.

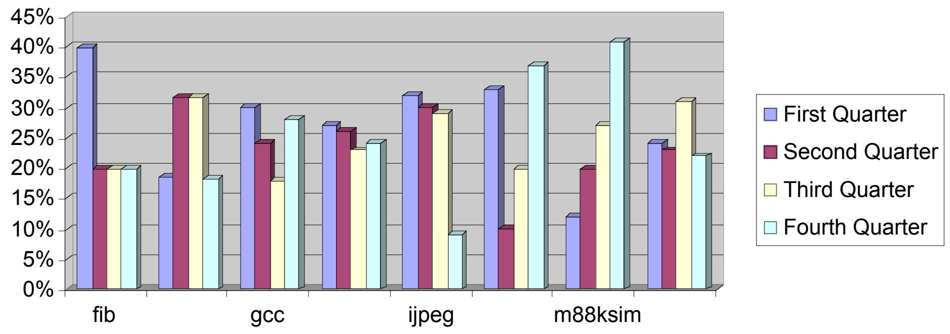


Figure 2. Function reuse at different execution times

The size and the organization of the reuse buffer plays a significant role in the performance gains that are possible. In the experiments thus far we assumed unlimited buffer sizes. In the next experiments we explored the impact of finite buffer sizes and associativity of the buffer on the possible function reuse. We used 128, 256, 512 and 1024 entry reuse buffers (with sufficient width to cache all input arguments for functions). In Figure 3, we show the result using direct-mapped reuse buffer. In Figure 4 we show results for 2-way set-associative reuse buffers. Figure 5 shows the data for fully associative reuse buffers. We use a modified Dinero cache simulator to obtain these results. As expected, higher reuse is achieved with larger buffers and higher set-associativities (compare the data in Figure 4 and Figure 5 with that of Figure 3) In order to optimize the use of the buffers to achieve better results, one needs to rely on compile time analyses and execution profiles so that only the functions that produce high reuse are cached.

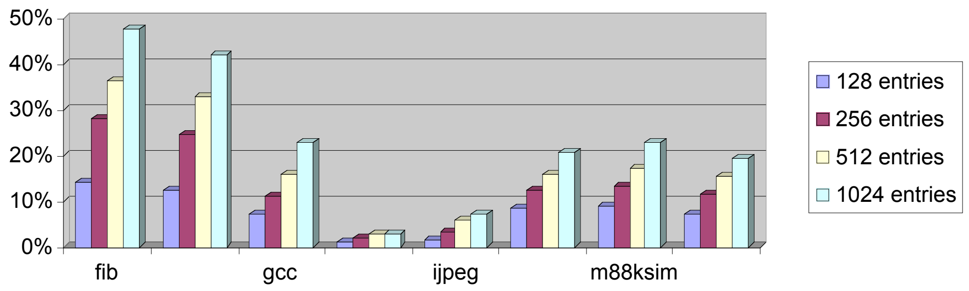


Figure 3. Function reuses for varying reuse buffer sizes

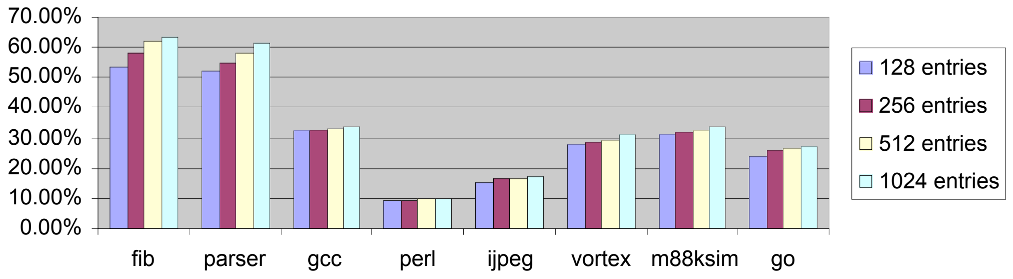


Figure 4. Function reuse for 2-way set-associative buffers

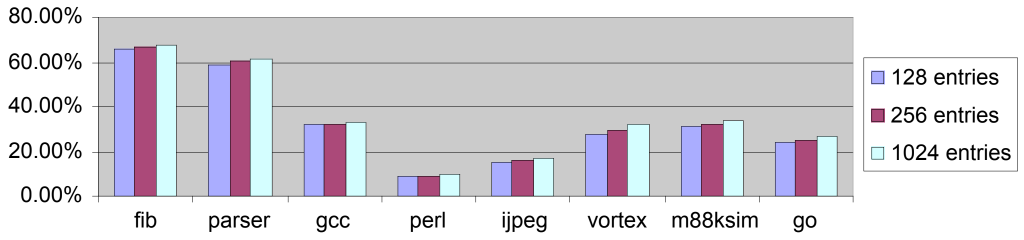


Figure 5. Function reuse for fully associative buffers

We were also curious how the reuse changes with the number of input arguments. It is intuitive to assume that functions with zero or one input have greater probability of being invoked with same input values than functions with higher number of arguments. In order to validate this belief, we explored the potential reuse based on the number of inputs. Figure 6 shows the data. The data is the aggregate for all benchmarks.

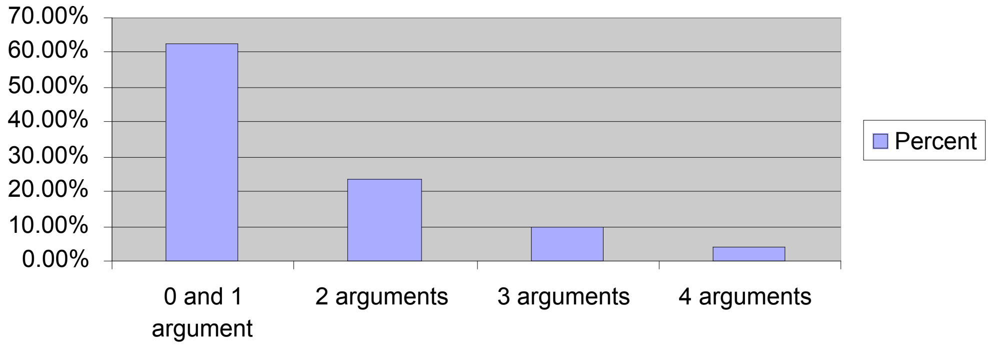


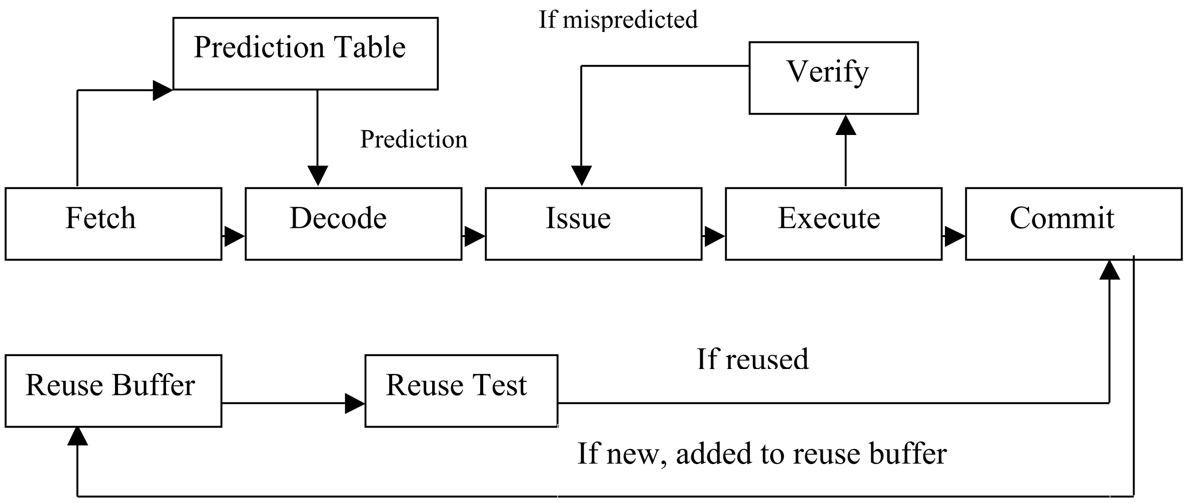
Figure 6. Function reuse for varying number of input arguments

As expected, functions with fewer arguments exhibited higher reuse. Such information is useful in deciding function values to cache. Moreover, fewer arguments will also lead to narrower reuse buffers since only few input values have to be remembered.

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 redundancy in programs by obtaining results of functions based on their prior executions, and

Figure 7. Function reuse architecture



thereby skipping repeated executions. Figure 7 shows a pipeline with function reuse. When a function is first executed, its results are stored in a hardware structure called a Reuse Buffer. The organization of the reuse buffer is shown in Figure 8. The buffer is indexed by the program counter of the function call (such as JSR). The reuse buffer is access 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-R4. When the 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 (R2 and R3 in MIPS). When this occurs, the function call is squashed and the program is continued from the instruction beyond the call (by simply incrementing the program counter). When function is not reused, the function with new arguments is cached in 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 8. Reuse buffer entry

6. Conclusions

In this paper we introduced and studied the concept of dynamic function reuse. 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. We also suspect that integer functions have higher likelihood of reuse than floating point functions. Our studies with integer benchmarks indicate the following

- There is a great potential for exploiting function reuse.
- The performance gains depend on the size and organization of the ruse buffer.
- Functions with fewer arguments have higher probability of reuse.

We believe that object-oriented programming promotes code reuse that 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 rely on compile time analyses so that that only most likely candidates for reuse are cached in the reuse buffer. Execution profiles may also be utilized to further improve the efficiency of the function reuse. We can further extend the idea to include function prediction (instead of reuse) in a multithreaded environment. We can continue the execution beyond a function call by spawning a thread using predicted values, and overlapping the execution of the function to verify the prediction.

Reference

[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] 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

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

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

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

[6] F. Gabbay and A. Mendelson; "Improving achievable ILP through value prediction and program profiling"; Microprocessors and Microsystems, Vol. 22. No. 6, Pages 315-332, November 30, 1998

[7] 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

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

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

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

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

[12] 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.

[13] 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.

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