H-0218 February 2004
Computer Science

IBM Research Report

Overlapping Memory Operations with Circuit Evaluation in Reconfigurable Computing

Yosi Ben-Asher
CS Haifa University
Israel

Daniel Citron, Gadi Haber
IBM Research Division
Haifa Research Laboratory
Haifa 31905
Israel

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T. J. Watson Research Center, P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home.

Overlapping Memory Operations with Circuit Evaluation in Reconfigurable Computing

Yosi Ben-Asher
CS Haifa University, Israel
yosi@cs.haifa.ac.il

Daniel Citron
IBM Research Labs in Haifa, Israel
citron@il.ibm.com

Gadi Haber
IBM Research Labs in Haifa, Israel
haber@il.ibm.com

Abstract

This paper considers the problem of compiling programs, written in a general high-level programming language, into hardware circuits executed by an FPGA (Field Programmable Gate Array) unit. In particular, we consider the problem of synthesizing nested loops that frequently access array elements stored in an external memory (outside the FPGA). We propose an aggressive compilation scheme, based on loop unrolling and code flattening techniques, where array references from/to the external memory are overlapped with uninterrupted hardware evaluation of the synthesized loop's circuit. We implemented a restricted programming language called DOL based on the proposed compilation scheme and our experimental results provide preliminary evidence that aggressive compilation can be used to compile large code segments into circuits, including overlapping of hardware operations and memory references.

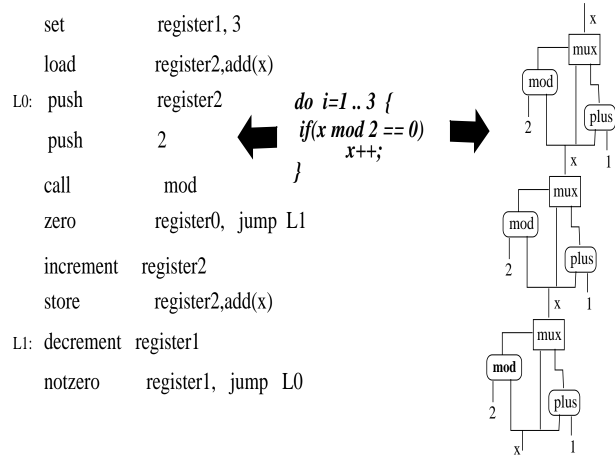

Figure 1. *Compiling into circuit versus compiling to machine instructions*

1 Introduction

Hardware compilation is an emerging technology used to compile programs or parts of programs directly to hardware circuits. Usually, the resulting circuits are executed on a reconfigurable device such as the FPGA, in order to accelerate execution. Hardware compilation is more economic in terms of power and execution time as there is no need to fetch values (from the memory/registers), decode or store. Ideally, values are directly propagated from one operation/gate to the other. Figure 1 demonstrates the approach of compiling a program directly into a hardware circuit (consisting of the $'+'$,$'mod'$ and $'mux'$ operations) vs. the usual method of compiling it into a fixed set of machine instructions. Hardware compilation can automatically expose hidden parallelism in the original code as a circuit is always evaluated in parallel (e.g., the mod and the $plus$ of the circuit in figure 1 are evaluated in parallel). Finally, (ideally), hardware compilation can ease the task of developing ASIC applications, allowing the developer to use high-level algorithmic description, rather than HDL languages such as VHDL.

Hardware compilation is a complex area of research that combines compilation techniques [14] with the study of various FPGA-based architectures [5]. In general, the circuits generated by the compiler are executed on an external reconfigurable unit (usually an FPGA co-processor) in three stages:

fetch - input values are loaded from the memory to the reconfigurable unit.

eval - the underlying circuit is evaluated in hardware.

update - the output values computed by the circuit are written back to the memory.

The overhead involved with data transfers between the reconfigurable unit and the memory can easily dominate the speedup obtained from the hardware evaluation of the circuit. For example, the GARP architecture [10] uses special hardware queues to speed up transfer of values between the memory and the reconfigurable unit. In this work we propose an aggressive compilation scheme that attempts to reduce this overhead, and thus improve the effectiveness of hardware compilation.

In this paper we describe the main requirements of a hardware compilation scheme needed in order to obtain efficient management of memory references by reconfigurable architectures. Consider the following basic model of hardware compilation:

Definition 1.1 *A code segment S of a program R is compiled to a generic circuit C_S that is evaluated whenever the execution of R passes through S. An evaluation of C_S is considered "efficient" if, when the execution of R passes through S, all the input values of C_S are available and ready to be fed into the circuit. Similarly, we require that the execution of R can continue immediately after the evaluation of C_S completes, without the need to wait for updating the memory with the values computed by C_S, to finish.*

Assume that both the code segment preceding S and the code executed immediately after S are compiled to circuits. Then the previous condition can be stated as follows:

Definition 1.2 *Let $S_1, \ldots, S_n$ be a sequence of code segments that are compiled to circuits $C_{S_1}, \ldots, C_{S_n}$ in respect and are executed one after the other. Then* $mem_ratio(S_1, \ldots, S_n) =$

$$\sum_{i=2}^{n-1} \frac{depth(C_{S_i})}{(load_in(C_{S_{i+1}}) + store_out(C_{S_{i-1}}))/2}$$

where $load_in(C_{S_i})$ is the number of C_{S_i}'s inputs that must be loaded from memory and $store_out(C_{S_i})$ are the outputs computed by C_{S_i} that must be stored to memory.

The mem_ratio is devided by 2 since the fetch and update operations can be performed in parallel.

Load/store operations can be overlapped with circuit evaluation only if $mem_ratio(S_1, \ldots, S_n) > 1$. The mem_ratio can also be computed for the size of C_{S_i} in which case the mem_ratio measure will serve as an upper bound to the amount of time available for memory operations.

The conditions for efficient hardware compilation can be stated as follows:

1. There is a partitioning of program R into code segments such that most of the code segments in the execution trace of R are compiled into circuits.
2. The mem ratio of any consecutive sequence $S_1, \ldots, S_n$ of code segments in the execution trace of R is greater than one.
3. All memory addresses in $load_in(C_{S_{i+1}})$ can be computed in hardware and in parallel during the evaluation of C_{S_i}.
4. All memory addresses in $store_out(C_{S_{i-1}})$ can be computed in hardware and in parallel during the evaluation of C_{S_i}.

Our proposed solution works by "flattening" each frequently executed code segment, containing one or several consecutive or nested loops, into a large "main-loop". The body of the main-loop (referred to here as the chunk) is compiled to a circuit such that each evaluation of this circuit computes one iteration of the main-loop. Each chunk may consist of several iterations of the original program loop (or loops) in the compiled code segment. For example, the loop

$$for\,(i = 0;\ i < n;\ i{+}{+})\,if\,(i\,mod\,5 == 0)\,x{+} = A[i{-}10];$$

can be executed in chunks of 10 iterations each. Thus, when computing variable x for the chunk of $i = 30, 31, \ldots, 39$, we can potentially load/prepare input values $A[30], A[35]$ that are needed for the next chunk of $i = 40, 41, \ldots, 49$.

Obtaining optimal partitioning to chunks such that the mem-ratio is larger than 1, is not always trivial. Consider for example the multiplication of two $n \times n$ matrices $C = A \times B$ with the inner loop:

$$for(k = 0; k < N; k + +)\ C[i][j] + = A[i][k] * B[k][j];$$

The resulting circuit is a "long" chain of $2n$ plus-gates and multiplications with about $2n$ load operations. Thus, for any partitioning to chunks, each chunk will include only long chains of the above type. Consequently, the mem-ratio of naive matrix multiplication is about 1 which is barely tolerated. However, compilers can overcome this problem by partitioning the A, B matrices into sub-matrices of size $Z \times Z$ each, and applying the matrix multiplication operation in the granularity of sub-matrices. Therefore, when

partitioning to chunks of size Z^3, the mem-ratio is $\frac{Z^3}{Z^2} = Z$, yielding potentially fast execution. These loop transformation techniques, suitable for improving the mem-ratio (including the above example) fall into the scope of existing compiler optimization methods aimed to improve data cache behavior.

In this work we developed a restricted programming language called DOL designed to test this hypothesis regarding the potential and the ability to overlap memory operations and circuit evaluation. We report preliminary results showing that programs written in DOL can achieve good memratio and can be efficiently compiled into hardware using the proposed scheme.

2 Related works

Related works can be divided into two categories: languages and compilation techniques, and reconfigurable architectures built to explore the premise of reconfigurable computing.

The basic concept of hardware compilation was proposed by N. Wirth [15], who showed how to synthesize high-level programs to circuits, based on their syntax tree. Currently, hardware compilation relies on modern compiler technology to synthesize code using "low-level" intermediate representations of programs (such as RTL, data dependency graphs, and control flow graphs). For example the PRISM-I/II compiler [1] works by first generating the RTL (Register Transfer Language) code, then generating circuits out of simple basic blocks using the data dependency graphs of these blocks.

Some hardware compilers work by extracting short instruction sequences and compiling them to circuits. These include the PRISC compiler [16] and the Chimaera C compiler [18]. Such compilers rely on combining techniques [14] to generate multi-operand instructions executed by the reconfigurable unit.

Some compilers require explicit statements to mark code segments that should be compiled to circuits, such as the Napa C compiler [9]. The GARP compiler [3] uses Instruction Level Parallelism (ILP) methods to compile the inner most loops of a C program to circuits. The GARP compiler uses hyperblock optimizations [12] to increase the size of the resulting circuits. There is also an attempt to use software pipelining [14] in the GARP compiler. Vectorization techniques have also been suggested for hardware compilation in [17]. Some efforts are made to apply hardware compilation on Java programs [18].

Another existing compilation direction is to use the C language together with parallel constructs to form high-

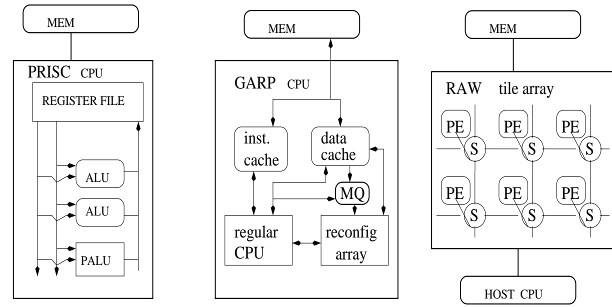

Figure 2. *Sample of reconfigurable architectures.*

level hardware description languages such as Transmogrifier C [8], Handel-C from Celoxica, and Streams-C.

There are many reconfigurable architectures of which we will describe three. Note that in these architectures, the reconfigurable unit must frequently access an external memory and fetch inputs for the circuit that is being evaluated by the reconfigurable unit. PRISK [16] (see figure 2) adds a programmable functional unit (PFU) to the datapath of the CPU. The PFU allows the CPU to execute special instructions generated by the PRISC compiler for the current program. Note that in this case, the reconfigurable unit works with the register file and not with the memory, thus reducing the special instructions to use only registers as inputs. The GARP architecture [10] (see Figure 2) uses a reconfigurable array as a co-processor to execute suitable inner loops of a program in hardware. Note that here, the reconfigurable array can access the data cache, the memory and the CPU registers, thereby speeding up load/store operations. The reconfigurable array of the GARP is a 2D array of 2-bit functional units that can be dynamically connected to form the circuits that are synthesized by the GARP compiler. The RAW machine [2] (see Figure 2) is a highly distributed architecture consisting of a 2D array of reconfigurable processing elements (PEs). The connections between the PES can be dynamically changed by the host. Thus, reconfiguration is executed both inside each PE, and between the PE themselves. Here, the memory is distributed, as each PE has its own memory but can also access other local memories using internal communication lines. Thus, the RAW machine has to suspend circuit evaluation to perform load/store operations.

3 Compilation method

As explained in the introduction, the proposed compilation scheme takes a sequence of nested do-loops and transforms them into one large loop (referred to as the"main-loop") whose body (called the "chunk") is compiled to a

combinatorial circuit (called the “generic circuit”). The goal is to show that such a partitioning strategy is (a) feasible and (b) obtains sufficiently high mem-ratios.

In general, the compiler uses several transformations (which can be regarded as special variants of known loop transformations [6]) to “flatten” the program to a main-loop. We omit the technical details and only demonstrate how the transformations work via examples (using C-like syntax):

Flattening two nested loops - This transformation is related to loop coalescing [6].

```
for(j = 0; j < n; j++) {
  c[i,j] = 0;
  for(k = 0; k < n; k++)
    c[i,j] += a[i,k] * b[k,j]; }
```

```
for(j = 0, k = 0; j < n;
  k < n ? k++ : j++; k = 0){
  if(k == 0) c[i,j] = 0;
  c[i,j] += a[i,k] * b[k,j]; }
```

Flattening two consecutive loops - This transformation is related to loop fusion [6].

```
for(k = 2; k < n; k++)
  a[i] = a[i-1] + a[i-2];
for(j = 0; j < n; j++)
  b[j] += a[i];
```

```
for(i = 0, j = 0;
  j < n; (i < n) ? i++ : j++)
  if(i < n) a[i] =
  a[i-1] + a[i-2];
  else b[j] += a[i];
```

Flattening conditional loop - If statements must be placed inside loops.

```
if(x < y)
  for(j = 0; j < n; j++)
    b[j] += a[i];
```

```
for(j = 0; j < n; j++)
{ int b = 0;
  if(i == 0 && x < y) b = 1;
  if(b == 1) b[j] += a[i]; }
```

Loop unrolling [14, 6] - used to increase the size of a loops body and consequently the size of the generic circuit. For example the code $for(k = 0; k < n; k++) c[i,j] += a[i,k] * b[k,j];$ is unrolled to

```
for(k = 0; k < n;){
 if(k + 3 < n){
  c[i,j] += a[i,k] * b[k,j];
  c[i,j] += a[i,k+1] * b[k+1,j];
  c[i,j] += a[i,k+2] * b[k+2,j];
  k += 3;  } else if(k == n-2){
  c[i,j] += a[i,k] * b[k,j];
  c[i,j] += a[i,k+1] * b[k+1,j];
  k = n
 } else if(k == n-1){
  c[i,j] += a[i,k] * b[k,j];
  k = n;
  }}
```

Only the innermost loops should be unrolled, as the body of a loop containing other loops cannot be compiled to a single combinatorial circuit (the generic circuit). This is true no matter how the inner loops are unrolled.

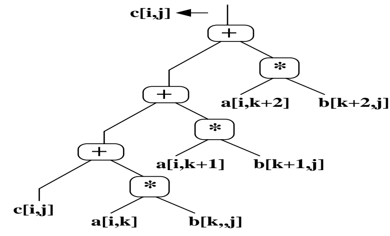

Figure 3. *Generic circuit of* $C[i,j]+ = A[i,k] * B[k,j]$.

The basic compilation technique is to unroll the innermost loops and then recursively (bottom-up) apply the transformations, until the whole code-segment has been flattened into one main loop.

The DOL compiler attempts to optimize the generic circuit by creating larger sequences of simple assignments that contain if-statements which slow down the resulted circuit. The if-statements are implemented in hardware by using special MUX-like circuits in order to reduce their overhead on execution time. Consider flattening a nested loop, assuming that the inner loop's range is relatively small compared to that of the outer loop. In this case, we would like to create sequences containing a maximal number of consecutive assignments uninterrupted by if-statements. This can be done by mixing statements from the last iteration of the inner loop and the following iteration of the outer loop.

The actual implementation of the above recursive scheme does not work directly on the abstract syntax tree of the program, but rather on an internal representation of the loops called the DOL representation. The DOL representation of loops is in fact a partition of the loop into cases, according to the possible values of the index variables of that loop. For example, assume that we wish to unroll the loop

$$for(k = 0; k < n; k++)\ c[i,j] += a[i,k] * b[k,j]$$

, $P = 3$ times. In loop unrolling, it is necessary to consider the possibility that the range n is not divided by P. The resulting partition to cases will therefore include the following cases: $k < n - 2$ (we are free to unroll three times),$k < n - 1$ (we unroll two times) and $k < n$ (last iteration is executed). Note that it is relatively easy to compile the condition/body/increment parts of each case to hardware, as these parts do not contain loops. For example, the body of the case $j < n - 2$ is depicted in Figure 3. In addition, computing the sequence of addresses and values needed by the load-in/store-out part of each case can be also generated using simple circuits.

Each application of a transformation results in a larger

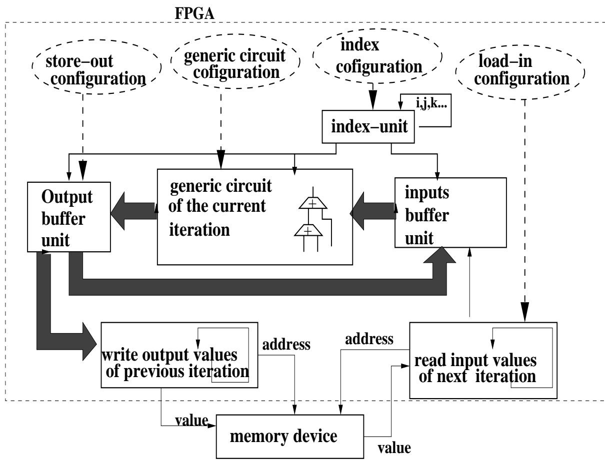

Figure 4. *Configuring an FPGA to evaluate DOL programs*

loop which is also represented by cases. Thus, the final main loop is also represented as a set of cases.

Our target architecture, as depicted in Figure 4, is an FPGA unit configured to execute this main loop as follows:

1. An external memory holds the arrays accessed by the program. We assume that the inputs to the program are the initial values of these arrays. We assume that the FPGA can load/store two items in parallel in about the same time needed to compute arithmetic operations (this is subject to using fast SRAM memories or using a cache).

2. The chunk of each case of the large loop is synthesized to a single circuit which is downloaded to a configuration cache of the FPGA. We thus assume that the FPGA can dynamically switch from one configuration to the other. Another alternative is to synthesize all the circuits of the cases to one “large” circuit, and use different parts of it when the execution needs to switch to a different case.

3. At each stage, the FPGA holds the values of the indexes $i, j, k, \ldots$ associated with the large loop.

4. The FPGA evaluates the condition expressions, and the suitable generic circuit is selected for the current iteration of the loop.

5. The set of items loaded from the memory in the previous iteration are loaded into the inputs of the current generic circuits from the inputs-buffer.

6. The evaluation of the generic circuits begins.

7. The set of addresses and values ($store - out$) of the previous iteration are written to the memory. This is performed while the generic circuit is evaluated, thus overlapping the evaluation time of the generic circuit with load/store operations.

8. The set of addresses $load - in$ of the next iteration of the large loop is computed, executed, and stored in the input buffer for the next iteration (overlapping evaluation times and load operations).

9. While the generic circuit is evaluated, the outputs-buffer is filled up with addresses and values that need to be stored in the memory.

10. Not every load/store operation need to be actually written to the main memory, in some cases, values can be directly fetched from the output circuit of the previous iteration.

11. The $increment$ expression of the current iteration is evaluated.

Though the above setting is a general one, we can draw some conclusions regarding necessary conditions so that efficient compilation to hardware can take place:

- The total number of cases after compilation must be a small constant, proportional to the number of loops in the source program. This is a key feature, allowing us to perform small number of “dynamic” reconfigurations at the FPGA, and preventing us from embedding too many generic circuits in the FPGA.

- On average (over the execution of the final chunk), the number of load-in/store-out operations should not exceed the average depth (in units of algebraic operations) of the generic circuit. This allows us to support the premise of the proposed approach, namely to overlap the evaluation time of the generic circuit, with the load/store of the inputs/outputs of the next/previous chunk.

- The addresses computed for the load-in/store-out units must be simple expressions of the loops’ indexes and induction variables (e.g., $a[i - 2]$ is preferable to $a[2 * i + j/\sqrt{i + j}]$). Simple indexing functions will allow the FPGA to compute the addresses not only in parallel, but perhaps also using no more than one or two arithmetic operations.

- The memory address of the load-in of any chunk must be such that it can be computed in parallel and before the chunk is evaluated. This may prevent us from using indirect expressions of the form $a[a[i]]+ = \ldots$ as the value of $a[a[i]]$ can be changed during evaluation of

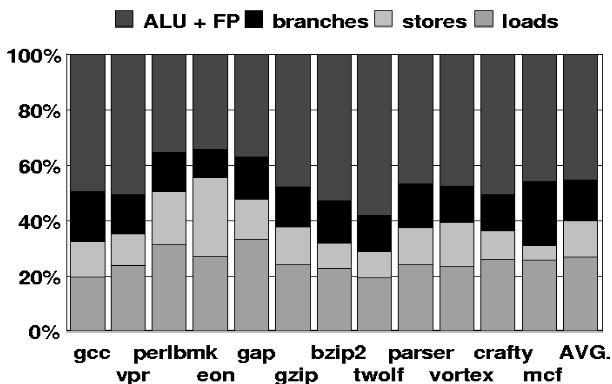

Figure 5. *Operations frequency in SPEC CINT2000*

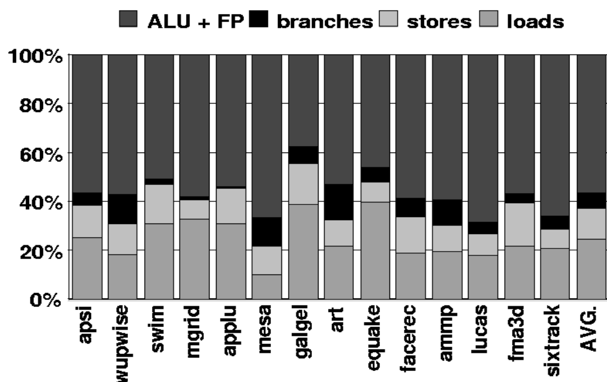

Figure 6. *Operations frequency in SPEC CFP2000*

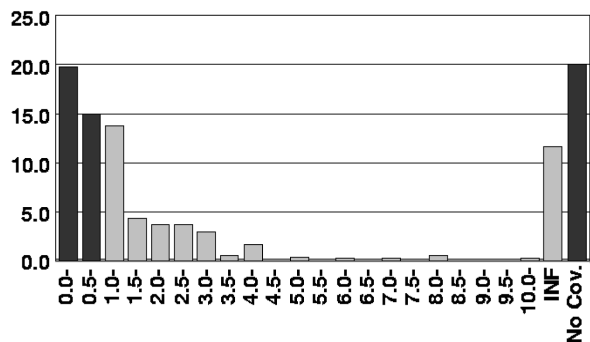

Figure 7. *Mem-ratio in SPEC INT 2000*

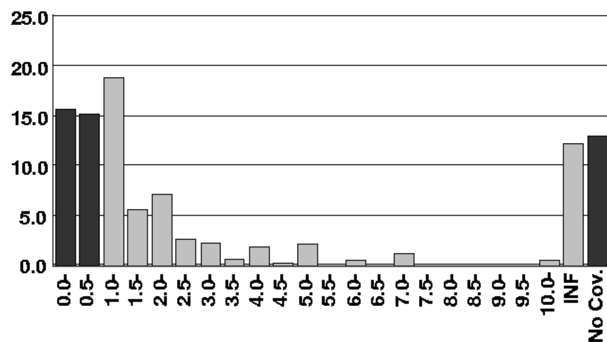

Figure 8. *Mem-ratio in SPEC FP 2000*

the current generic chunk. For such cases, we are able to provide only some restricted solutions, as will be explained in the next section.

4 Experimental results

The first set of experiments was designed to check the mem-ratio in general programs. We used a post-link tool to count the total number of execution of each type of instruction (load/store/branch/ALU+FP) in the SPEC CPU 2000 benchmark suite running on the 64bit IBM Power3 architecture. The results in Figure 5 for SpecInt2000 and in Figure 6 for SpecFP2000 support the fact that the mem-ratio of $\frac{alu}{(load+store)/2}$ is about 3. Similar results were also obtained by other works [11, 4] for SPEC CPU 2000.

In the second set of experiments we used the ARIA simulator [13] that was modified to check the mem-ratio in "traces" that are collections of basic blocks. We stop building a trace when the number of instructions is larger than 10 or a backwards branch or branch to a register value is encountered. A trace that is smaller than 4 instructions is not counted. Figures 7 and 8 show histograms of the mem-ratio. The rightmost bars are the traces that didn't have any memory references and the traces that were smaller than 4 instructions. The results show that about 50% of the traces have a mem-ratio higher than one. In fact, in these charts the mem-ratio was not divided by two so that the actual result is even higher. Note that working with traces of size less than 10 does not reveal the full potential of the proposed scheme as the underlying compiler did not "flattened" loops and the traces are only an approximation to fully "flattened" loops.

We developed a special programming language called DOL (Do-loops Language) that enables the aggressive translation of programs to circuits, based on the compilation concept described in Section 3. DOL was designed to provide a preliminary proof of concept for the compilation scheme described in Section 3 by showing that flattening loops can be used to synthesize circuits. By using the DOL compiler we were able to measure the ratio between memory load/stores instructions and internal hardware operations of in the generated circuits.

We have implemented several algorithms in DOL lan-

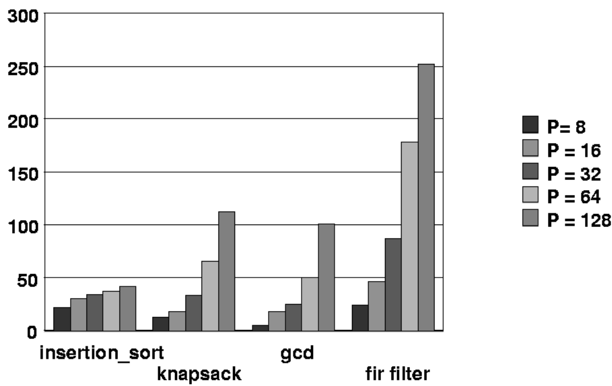

Figure 9. *mem-ratio (size/(load,store)) for simple DOL programs.*

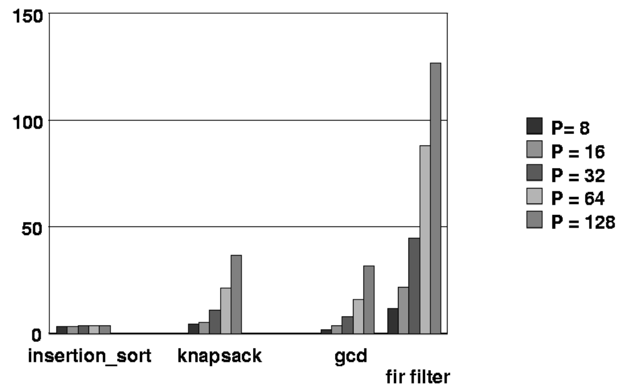

Figure 10. *mem-ratio (depth/(load,store)) for simple DOL programs.*

guage. No optimizations were applied on the implemented algorithms for improved mem-ratio behavior. Figure 9 describes the average mem-ratio size $\frac{circuit_size}{(load+stores)/2}$ for the unrolling factors $P = 8 \ldots 128$. Note that the value of P also determines the expected size/depth of the generic circuit, e.g., for $P = 16$, the resulting circuit will have twice the size/depth of the circuit obtained for $P = 8$. As can be seen from the results, large factors of mem-ratio have been obtained for programs written in DOL. Moreover, the mem-ratio grows proportionally to P, hence any desired mem-ratio can be obtained by selecting the right value of P. This follows from the fact that as the unrolling factor increases, more values that were computed in previous iterations of the main-loop, can be directly used as inputs for the next iterations. A sufficiently good mem-ratio has also been obtained for the depth of the generic circuit (Figure 10). A high mem-ratio for the depth reduces the evaluation time of the generic circuit to the minimum possible by the critical path length of the original code. This follows from the fact that the proposed compilation scheme does not insert dependencies which were not present in the original code.

Following are the results of applying DOL to the well-known Livermore loops. This set of 24 sequential loops (extracted from FORTRAN numerical code) has been used to evaluate the power of parallelizers and parallel systems to extract parallelism out of sequential code since the early seventies. As described in [7] not every loop can be parallelized and some are harder to synthesize than others. Loops which result in DOL programs that include double referencing (loops 2,8,13,14,15,16,17,18,24) are obviously not included (though some cases can be handled using special features of DOL). The mem-ratio for both, size and depth (see Figures 11, 12) can be divided into two groups: loops $1, 9, 7, 8, 12$ with low mem-ratio of less than 1, and the remaining loops with a mem-ratio greater than 1. The first

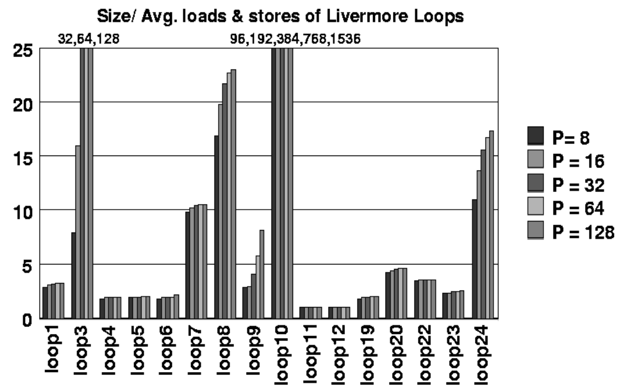

Figure 11. *Livermore loops mem-ratio for size.*

group (loops $1, 9, 7, 8, 12$) are highly parallel loops, where each array element is updated only once during the loop iterations by values which are not computed within the loop. Clearly, such cases (e.g., $for(i = 0; i < n; i++)\ a[i] = b[i] + c[i]$) cannot obtain good mem-ratio, no matter how we rewrite the loop. However, such initializing loops cannot play a significant part in complex applications. Note that for these loops, increasing P reduces the mem-ratio (rather than improving it). This is because the depth of their resulted generic circuits remains constant for every P. Therefore, selecting a higher P implies that relatively more store/load operations occur for every ALU/FP operation.

Though these experiments do not include full benchmarks or large applications, they do demonstrate the premise of the proposed method, namely that programs or special loops which do not include multiple double references can be efficiently compiled into hardware.

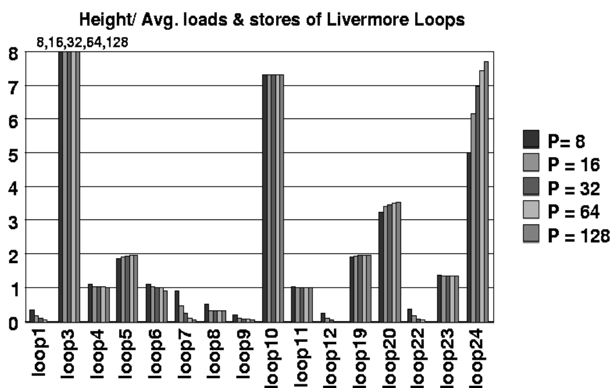

Figure 12. *Livermore loops mem-ratio for depth.*

5 Conclusions

In this work, we have described a hardware-compilation scheme of programs to circuits. The basic idea is to compile a program (or a part of it) into a single main loop whose body does not contain any branch instructions. Consequently, the body of the main loop can be compiled to a combinatorial circuit, called the generic circuit. The goal is to execute the main loop by repeated uninterrupted evaluation of the generic circuit. This can be done only if we are able to fetch input values for the next evaluation of the generic circuit, while we are evaluating the current one. We have shown that this can be done if the "mem-ratio" of the main loop's body is greater than one. We have implemented a simple programming language called DOL in order to test this assumption. Though the experiments presented here are preliminary, they do support this assumption.

References

[1] P. M. Athanas. A functional reconfigurable architecture and compiler for adaptive computing. In *12th Annual International Phoenix Conference on Computers and Communication*, pages 49–55, 1993.

[2] J. Babb, M. Frank, E. Waingold, R. Barua, M. Taylor, J. Kim, S. Devabhaktuni, and A. Agarwal. The RAW benchmark suite: Computation structures for general purpose computing. In K. L. Pocek and J. Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 134–143, Los Alamitos, CA, 1997. IEEE Computer Society Press.

[3] T. J. Callahan and J. Wawrzynek. Instruction-level parallelism for reconfigurable computing. In R. W. Hartenstein and A. Keevallik, editors, *Field-Programmable Logic: From FPGAs to Computing Paradigm*, pages 248–257. Springer-Verlag, Berlin, / 1998.

[4] J. F. Cantin and M. D. Hill. Cache performance for spec cpu2000 benchmarks.

[5] K. Compton and S. Hauck. Reconfigurable computing: A survey of systems and software. *ACM Computing Surveys*, 2000.

[6] O. J. S. David F. Bacon, Susan L. Graham. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, 1994.

[7] J. T. Feo. An analysis of the computational and parallel complexity of the livermore loops. *Parallel Computing*, (7):163–185, 1986.

[8] D. Galloway. The transmogrifier C hardware description language and compiler for FPGAs. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 136–144, Napa, CA, Apr. 1995.

[9] M. B. Gokhale and J. M. Stone. NAPA C: Compiling for a hybrid RISC/FPGA architecture. In K. L. Pocek and J. M. Arnold, editors, *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '98)*, pages 126–135. IEEE Computer Society, IEEE Computer Society Press, Apr. 1998.

[10] J. R. Hauser and J. Wawrzynek. Garp: A MIPS processor with a reconfigurable coprocessor. In K. L. Pocek and J. Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 12–21, Los Alamitos, CA, 1997. IEEE Computer Society Press.

[11] A. KleinOsowski and D. J. Lilja. Adapting the spec 2000 benchmark suite for simulation-based computer architecture research. *Computer Architecture Letters*, 1, 2002.

[12] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *25th Annual International Symposium on Microarchitecture*, 1992.

[13] M. Moudgill, J. Wellman, and J. Moreno. Environment for PowerPC Microarchitecture Exploration. *IEEE Micro*, 19(3):15–25, May/June 1999.

[14] S. Muchnik. *Advanced Compiler Design Implementation*. Morgan Kaufmann, 1997.

[15] K. V. Palem, S. Talla, and W.-F. Wong. Hardware compilation: Translating programs into circuits. *IEEE Computer*, 31:25–31, 98.

[16] R. Razdan and M. D. Smith. A high-performance microarchitecture with hardware-programmable functional units. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 172–80. IEEE/ACM, Nov. 1994.

[17] M. Weinhardt and W. Luk. Pipeline vectorization for reconfigurable systems. In K. L. Pocek and J. M. Arnold, editors, *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, page n/a, Napa, CA, Apr. 1999. IEEE Computer Society, IEEE.

[18] Z. A. Ye, N. Shenoy, and P. Banerjee. A c compiler for a processor with a reconfigurable functional unit. In *ACM International Symposium on FieldProgrammable Gate Arrays*, pages 95–100, 2000.