'Towards automated extraction and hardware acceleration of performance critical program blocks – Preliminary experiments with SPEC benchmark suite '

Written by: Ali Zaidi

Supervisor: Dr. Oskar Mencer

Abstract

Acceleration of 'performance-critical' program blocks implemented in software by using reconfigurable devices has been a subject of considerable research in recent times. This interest can primarily be credited to the consistent advances in the performance and features of reconfigurable devices culminating in a 'technological threshold' followed by a general realization of their potential in High Performance Computing (HPC) applications. After careful analysis of the forefront of these developments, it can be stated that a vast majority of these efforts rely upon manual extraction of core prior implementation in hardware. Needless to say, an automatic and optimal extraction of the core can be of great benefit. In order to achieve this goal, we need to develop a thorough understanding of the problem and the solution space. In this ISO, we will study the SPEC benchmark suite and attempt to identify and thus extract the respective cores using two different strategies. We will estimate the potential speed-up obtained by implementing these cores on a reconfigurable device and examine the various parameters which affect the speed-up figure. Finally, we will compare the results from the two extraction strategies in order to identify the respective strengths and limitations.

Acknowledgements

I would like to acknowledge the contribution of Kubilay Atasu and Carlos Tavarez during this work. As always, Dr. Oskar Mencer was ever present with his advise when needed.

Table of Contents

Chapter 1 Introduction

The primary methods employed in contemporary computing applications for the implementation and/or acceleration of algorithms can be classified into three categories:

1. Implementations using 'Hard-Wired' devices
2. Implementations using 'Programmable' microprocessors
3. Implementations using 'Reconfigurable' devices

1.1 Implementations using 'Hard-Wired' devices

The first method employs 'Hard-Wired' devices for the implementation of algorithms. These devices include Application Specific Integrated Circuits (ASICs), configured to perform a certain computation, or several off-the-shelf, Large scale integration (LSI) or Medium Scale Integration (MSI) devices, connected on a Printed Circuit Board (PCB) to perform the required operations. Of these two, ASICs delivers better performance. This is largely due to the fact that the signal propagation delays are significantly smaller for on-chip connections as compared to PCB connections. However, the ASIC is prone to incur higher Non-Recurring Engineering (NRE) costs. Both of these solutions have a slight disadvantage when it comes to flexibility. In case of any design changes or upgrades, the cost for replacing the ASIC or redesigning the PCB is a significant factor which weighs against these solutions.

1.2 Implementations using programmable devices

The second method for the execution of algorithms is to use 'Programmable' microprocessors. The immediate advantage is that of flexibility. Microprocessors execute a set of instructions to perform the requisite computations. Any change in functionality of this computation can be easily incorporated by changing the set of instructions by reprogramming the device. However, there is a certain price to pay for this flexibility in terms of device performance and throughput. This is due to the fact that the very nature of the microprocessor allows for a sequential execution of tasks. The microprocessor must read each instruction from the memory, decode it, and only then executes it. The execution phase contributes towards the overall computational goal, while the fetch and decode cycles for every instruction can be termed as execution overheads. Additionally, the instruction set of a microprocessor is determined during the architectural design phase. Therefore, any attempt towards implementing an algorithm is limited by the instruction set for the particular microprocessor, although the control for the flow of instructions to be executed rests with the programmer.

1.3 Implementations using Reconfigurable devices

The diverse advantages and disadvantages of the implementation of computational algorithms using hard-wired and programmable solutions have resulted in a technology gap between the two domains. The last decade have witnessed the emergence of 'Reconfigurable' devices, which promise to bridge the gap between the traditional solutions by matching the performance of ASICs and being as flexible as microprocessors. Although several devices are included in the family of reconfigurable devices, however, Field Programmable Gate Arrays (FPGAs) are claimed by many to be the flag bearers of the potential of reconfigurable logic devices and generally assert the technological trends which shape the industry.

Reconfigurable devices, comprises of generic arrays of computational elements termed as 'Logic Blocks' whose functionality is determined by a configuration bit stream. The logic blocks are connected by sets of routing resources. These routing resources are implemented by on-chip 'Crossbar Switch Matrices' which, in turn are also configurable. Computational algorithms can be realized by dividing them into logic components which can be mapped directly on to the logic blocks. Subsequently, the logic blocks contributing towards the computational algorithm are connected by the configurable crossbar switch matrices to form the necessary circuit.

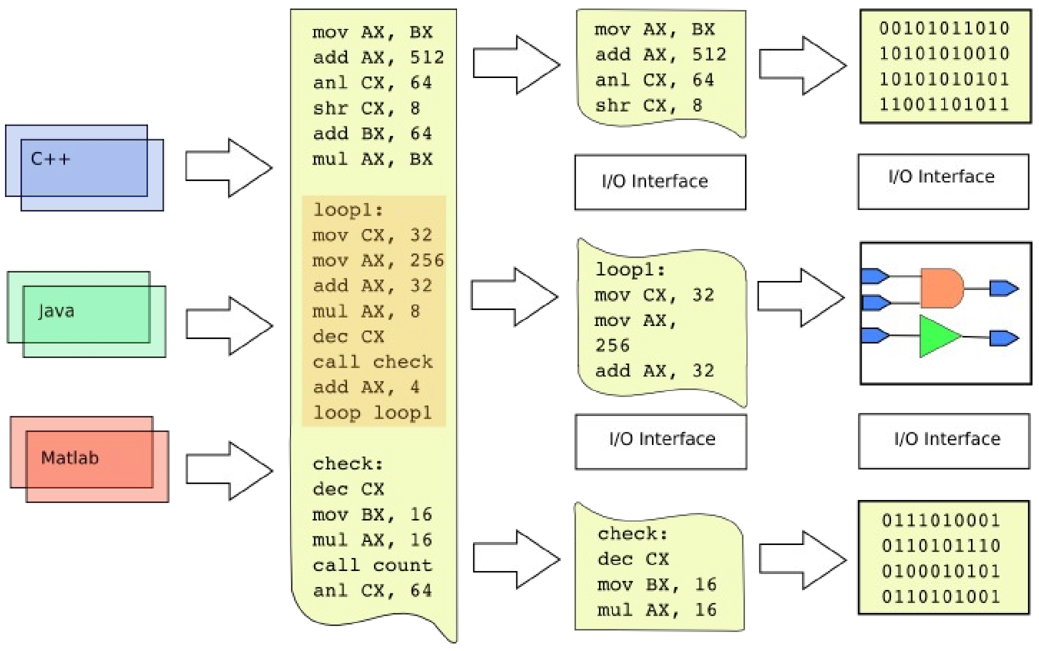
It can be stated that the implementation of computational algorithms in configurable devices is accomplished in 'Space Domain'. On the other hand, the implementation of computational algorithms in microprocessors are executed in 'Time Domain'. Unlike the microprocessors which broadcast instructions to the functional units on every clock cycle, instructions in reconfigurable devices are locally configured, allowing the reconfigurable device to compress instruction stream distribution and thus deliver more instructions into active silicon on each cycle. Reconfigurable devices provide a large number of separately programmable, relatively small computational units allowing for the execution of a greater range of computations per unit time. Resources such as memories, crossbar switch matrices and logic blocks are distributed rather than being centralized in large pools. Independent, local access allows for the utilization of on-chip resources in parallel, thus avoiding performance bottlenecks resulting from a large, central resource pool.

FPGAs and reconfigurable computing have been shown to accelerate a variety of applications. By careful analysis of these works, we can categorize the circumstances in which the traditional microprocessors or the contemporary reconfigurable devices are preferable. For example, when the function and data granularity to be computed are well-understood and fixed, and when the function can be economically implemented in space, dedicated hardware provides the most computational capacity per unit area to the application. On the other hand, if we are limited spatially and the function to be computed has a high operation and data diversity, we are forced into reusing limited active space and accepting limited instruction and data bandwidth. In this case, conventional microprocessors are most effective since they dedicate considerable space to on-chip instruction storage in order to minimize off-chip instruction traffic while executing descriptively complex tasks.

1.4 The 'Fourth Dimension'

Recently, we have seen the emergence of Reconfigurable devices in general and and FPGA in particular for accelerating 'performance critical' program blocks of computationally intensive algorithms primarily implemented in software. These developments can primarily be credited to the consistent advances in the performance and features of reconfigurable devices culminating in a 'technological threshold' followed by a general realization of their potential in High Performance Computing (HPC) applications. Hybrid implementation of computationally intensive algorithms have the potential to provide the best of both worlds. The control portions of the algorithm call for microprocessor implementation while the computationally expensive 'performance critical' portions of the algorithm should benefit from the custom hardware options provided by FPGA as shown in Figure-1.1.

The implementation of the algorithm on hybrid platform must take into account the time to transfer the data to be processed to the reconfigurable device as well as the time to accumulate the results. These timings can have a significant effect on determining the feasibility of the hybrid implementation as shown in Figure-1.2.



```
mov AX, BX
add AX, 512
anl CX, 64
shr CX, 8
add BX, 64
mul AX, BX

loop1:
mov CX, 32
mov AX, 256
add AX, 32
mul AX, 8
dec CX
call check
add AX, 4
loop loop1

check:
dec CX
mov BX, 16
mul AX, 16
call count
anl CX, 64
```

```
mov AX, BX
add AX, 512
anl CX, 64
shr CX, 8
```

```
loop1:
mov CX, 32
mov AX,
256
add AX, 32
```

```
check:
dec CX
mov BX, 16
mul AX, 16
```

```
00101011010
10101010010
10101010101
11001101011
```

```
0111010001
0110101110
0100010101
0110101001
```

Figure -1.1 An illustration of the extraction of performance critical program blocks form software and their implementation on hardware platform. Notice that the source code can be in any programming language. This is possible because the program blocks are being extracted at the assembly level. Notice that the execution time of the final implementation depends upon the bandwidth of the I/O interface.

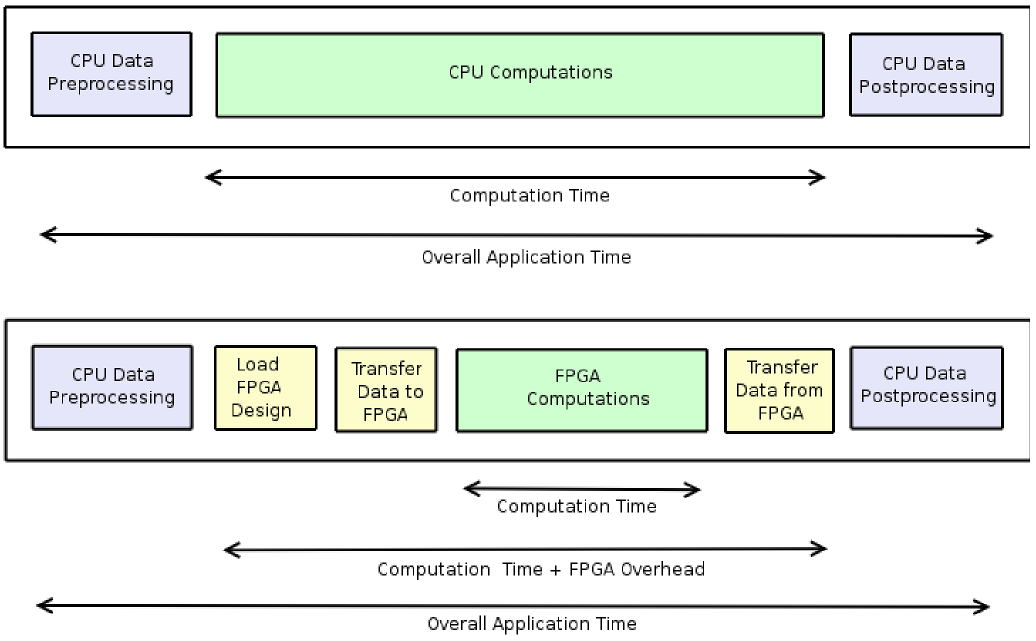


Figure -1.2 Comparison of execution times for CPU and hybrid implementations. Hybrid implementation generally reduces the computation time but results in communication overheads due transfer of data to/from FPGA.

After careful analysis of the forefront of these developments, it can be stated that a vast majority of these efforts rely upon manual extraction of 'performance-critical' program blocks which may benefit from implementation in hardware. An extremely beneficial direction for work can be the automated detection of the requirement of hardware acceleration in an algorithm, followed by the automated extraction and subsequent implementation of the critical areas of the algorithm on to reconfigurable hardware for algorithm acceleration. This development will result in generic accelerator cards available for immediate application independent deployment for the execution of 'performance critical' areas of program code.

In order to achieve this goal, we need to develop a thorough understanding of the problem and the solution space. In this ISO, we will study the SPEC benchmark suite and attempt to identify and thus extract the respective 'performance critical' program blocks, hereafter called the 'execution cores' or simply 'cores', using various strategies. We will calculate the potential speed-up obtained by implementing these cores on a reconfigurable device. Finally, we will compare the results from various extraction strategies to gain valuable insight into the solution space.

1.5 SPEC95 Benchmark Suite

The benchmarks suite chosen for this study is the SPEC95 benchmark suite. The various benchmarks of SPEC95 suite can be categorized as legacy code and therefore, they cannot gain any benefits from contemporary microprocessor features for high performance computing such as Superscalar/Vector processing support such as Intel SSE Extensions and Multiple Cores. Moreover, these software are not optimized for execution on modern microprocessors architecture extensions. They were designed to run with much smaller caches and memory. Such legacy programs represent a distinct advantage of using reconfigurable hardware accelerators. The critical loops inside these programs can be executed on hardware to gain significant speed-ups.

We have employed the CFP95 subset of the SPEC95 benchmark suite. Our selection can be justified by the fact that the reconfigurable hardware accelerator is targeted towards scientific and multimedia applications which benefit from vectorization.

A brief summary of the respective CFP benchmarks along with some control flow parameters such as branch ratio and ticks per block entry is provided below. Branch ratio is defined as the number of taken branches divided by the number of not-taken branches. A high value of the branch ratio will indicate that the program has many loops. Ticks per block entry is defined as the average time spent in each program block. It gives an idea of how large the program blocks are.

1.5.1 Tomcatv

Tomcatv is a double precision floating point mesh generation program which does little I/O and is described to be 90 - 98 % vectorizable. The various control flow statistics of tomcatv are shown in Figure-1.3 and Figure-1.4. Following are some of the important observations regarding the statistics:

- tomcatv comprise a single main function doing all the computing. The core comprise of a nested loop computing the residuals among a few other loops.
- It has a relatively modest branch ratio of 2.6 which signifies that this benchmark has a good few loops.
- The average ticks per block entry figure is the highest among CFP95 benchmarks – 130. This means that it has computationally large program blocks which are good for streaming.

1.5.2 Swim

Swim is a scientific benchmark with single precision floating point arithmetic. SWIM stands for Shallow Water Model with 1024 x 1024 grid. The program solves the system of shallow water equations using finite difference approximations on a N1 x N2 grid. The various control flow statistics of swim are shown in Figure-1.3 and Figure-1.4. Following are some of the important observations regarding the statistics:

- The swim core comprise three functions named calc1,calc2 and calc3 which account for 33, 25 and 25 percent of the total execution time respectively.
- The branch ratio of swim is 511 which is among the highest in CFP95 suite and shows that there are quite a few loops with large number of iterations.
- The ticks per block figure is fairly large – 73. This shows that the respective program blocks have a fair amount of computation.

1.5.3 Su2cor

The program is a vectorizable with double precision floating-point arithmetic. It is a quantum physics application, masses of elementary particles are computed in the framework of the Quark Gluon theory. The data are computed with a Monte Carlo method taken over from statistical mechanics. The various control flow statistics of su2cor are shown in Figure-1.3 and Figure-1.4. Following are some of the important observations regarding the statistics:

- The su2cor core comprises two functions named matmat and int2v.
- These functions account for 30 and 28 percent of the total program execution time respectively.
- The branch ratio for su2cor is approximately 16 which signifies a fair amount of loops.
- Average number of ticks per program block is 82.

1.5.4 hydro2d

The program is a vectorizable with double precision floating-point arithmetic. It is an astrophysics application which solved hydrodynamical Navier Stokes equations to compute galactical jets. The various control flow statistics of hydro2d are shown in Figure-1.3 and Figure-1.4. Following are some of the important observations regarding the statistics:

- The hydro2d core comprise of a single function named filter.
- This function accounts for about 42 percent of the total execution time. This means that su2cor has a good core element.
- The branch ratio for hydro2d is 9.
- Average number of ticks per program block entered is approximately 56, which is modest.

1.5.5 mgrid

mgrid is a multi-grid solver for 3D potential grid. mgrid demonstrates the capabilities of a very simple multi-grid solver in computing a three dimensional potential field. Adapted by SPEC from the NAS Parallel Benchmarks with modifications for portability and a different workload. The various control flow statistics of mgrid are shown in Figure-1.3 and Figure-1.4. Following are some of the important observations regarding the statistics:

- The mgrid core comprise two functions named resid and psinv.
- These functions account for 56 and 24 percent of the total program execution time respectively.
- Branch ratio for mgrid is 35 which signifies a large number of frequent loop iterations.
- Average number of ticks per program block entered is 72.

1.5.6 applu

applu is a Computational Fluid Dynamics and Computational Physics benchmark. Solution of five coupled non-linear PDE's, on a 3-dimensional logically structured grid, using an implicit pseudo-time marching scheme, based on two-factor approximate factorization of the sparse Jacobian matrix. The various control flow statistics of applu are shown in Figure-1.3 and Figure-1.4.

Following are some of the important observations regarding the statistics:

- The applu core comprises three functions named buts, blts and jacld.
- These functions account for 31, 25 and 17 percent of the total execution time respectively.
- The branch ratio is fairly small – 3
- Average ticks per block entry for this benchmark is 59.

1.5.7 turb3d

Turb3d simulates isotropic, homogeneous turbulence in a cube with periodic boundary conditions in x,y,z coordinate directions. It has a large 1D FFT computational component. It solves the Navier-Stokes equations using a pseudo spectral method. Leapfrog-Crank-Nicolson scheme is used for time stepping. The various control flow statistics of turb3d are shown in Figure-1.3 and Figure-1.4. Following are some of the important observations regarding the statistics:

- The turb3d core comprises three methods named fftz2, dcft and fftz1.
- These functions account for 30, 22 and 17 percent of the total program execution time respectively.
- The branch ratio for turb3d is 5.71.
- Average ticks per block entry for this benchmark is 52.

1.5.8 apsi

apsi is a scientific benchmark with double precision floating point arithmetic. It solves for the mesoscale and synoptic variations of potential temperature, U AND V wind components, and the mesoscale vertical velocity W pressure and distribution of pollutants C having sources Q. The synoptic scale components are in quasi-steady state balance, while the mesoscale pressure and velocity W are found diagnostically. The various control flow statistics of apsi are shown in Figure-1.3 and Figure-1.4. Following are some of the important observations regarding the statistics:

- The apsi benchmark has the most evenly distributed computational core in the CFP95 suite comprising of three functions named trid, radb4 and rad4.
- These functions account for approximately 14, 10 and 7 percent of the total execution time of the program respectively.
- The branch ratio is 9.
- Average number of instructions per block entry is 59.

1.5.9 fppp

A scientific benchmark with double precision floating point arithmetic. Fpppp is a quantum chemistry benchmark which measures performance on one style of computation (two electron integral derivative) which occurs in the Gaussian series of programs. It does very little I/O. The input to the program is found in a file and contains as the first entry, the number of atoms. The various control flow statistics of fpppp are shown in Figure-1.3 and Figure-1.4.

Following are some of the important observations regarding the statistics:

- The fpppp core is highly concentrated. It comprises of two functions named twldrv and fpppp.
- These functions account for 59 and 31 percent of the total program execution time respectively.
- Branch ratio for fpppp is 0.94 which is the lowest among the CFP95 benchmarks and shows and shows that the fpppp benchmark contains very few loops.
- Average number of instructions per block entry for the fpppp benchmark is 94.

1.5.10 wave5

Scientific benchmark with double precision floating point arithmetic. A two-dimensional, relativistic, electromagnetic particle-in-cell simulation code used to study various plasma phenomena. WAVE solves Maxwell's equations and particle equations of motion on a Cartesian mesh with a variety of field and particle boundary conditions. The benchmark problem involves 750,000 particles on 75,000 grid points for 40 time steps; about 11 M words (32-bit) of memory are required. Considerable indirect addressing dominates the code's runtime. The various control flow statistics of wave5 are shown in Figure-1.3 and Figure-1.4. Following are some of the important observations regarding the statistics:

- The wave5 core comprises of a single function called parmvr.
- This function accounts for 54 percent of the total program execution time signifying a highly concentrated core.
- Branch ratio for the wave5 benchmark is 30 which signifies a large number of loops and iterations.
- Average number of instructions per program block entered is 56.

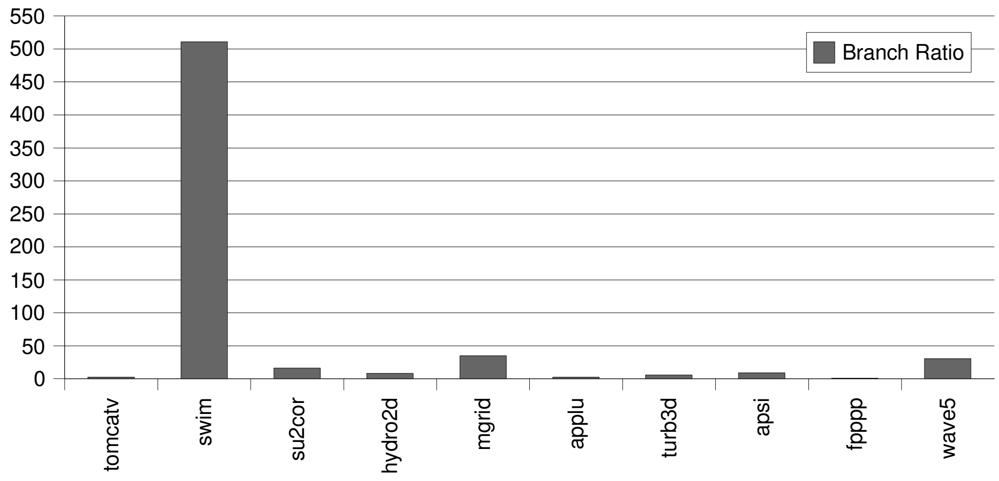


Figure-1.3 Branch ratio of the CFP95 suite. Branch ratio is the ratio between taken and not-taken branches during program execution.

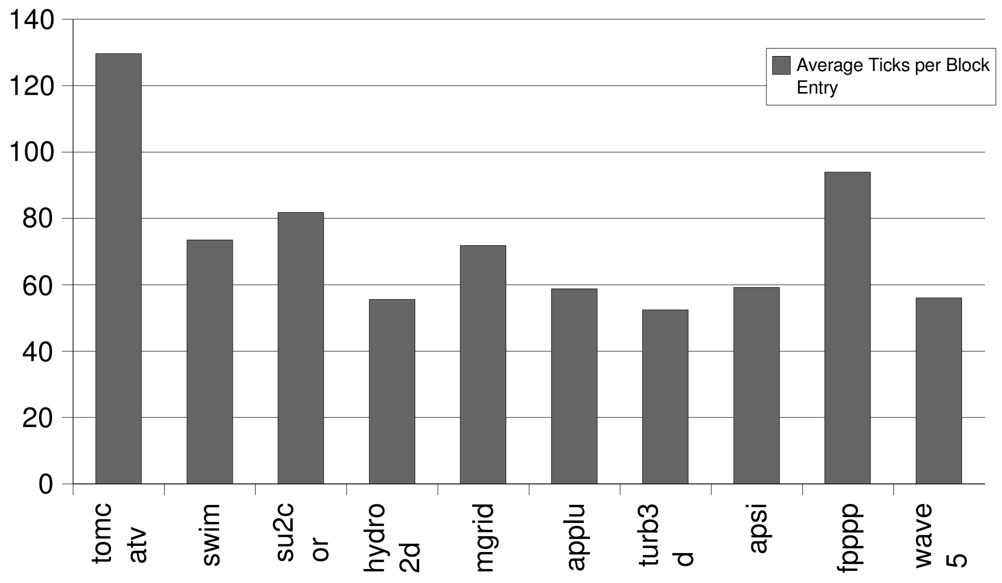


Figure-1.4 Average ticks per block entry for the CFP95 suite. Average ticks per block entry is a measure of the average size of the program blocks.

1.6 3S Program Instrumentation Framework

3S stands for Spacey Stream Splitter. It has a low overhead and the capability to produce the same type of reports as Valgrind. 3S framework is designed to instrument x86 assembly programs. It adds instrumentation stubs in the x86 assembly. These instrumentation stubs communicate with a library of 3S tools to pass them useful runtime information about program flow and other useful parameters such as the number of ticks the last block took to execute.

The instrumentation stubs can be placed around the basic program blocks such as loops and calls. In addition, it can also be placed around every instruction. The former is called program block level instrumentation while the latter is known as instruction level instrumentation. The 3S tools build upon the framework to produce various types of useful reports by employing the information they communicated by the 3S framework.

The 3S framework works at the assembly level. This way, 3S does not have to be concerned with its stubs overwriting program instructions or jump targets. In addition it benefits from the basic block identification algorithm already implemented in the compiler. Instrumenting at the assembly level is a parsing process which involves identifying the basic program blocks and adding instrumentation stubs around these blocks. Another advantage of working at the assembly level is that the output of the 3S framework is a fully readable assembly code. This adds transparency to the whole procedure of instrumentation. The users can actually see the bits added by the framework.

Following is a brief description of the various 3S block level instrumentation tools:

- **Hotspot:** Records the number of ticks and instructions for each block executed.
- **Trace:** Blindly writes the name of each block out to a file as it is encountered in the program.
- **Branch:** Records results of any branches that occur from the program trace.
- **ControlFlow:** Produces a control graph in pdf colouring the nodes in the graph according to how much time is spent in the corresponding block.
- **Fpr02_cfg:** Alternative control flow graph, merges sequential blocks into single nodes.
- **InstructionCount:** Records the category of instructions executed dynamically.
- **InstructionCountBoundary:** Records the category of instructions executed only within the boundary defined by FPGA markers in the original source code.
- **Loopgraph_d:** Records the program trace, similar to regex, but produces a pdf graph to illustrate the loops within the trace. Does not handle branching in code.
- **None:** Empty tool. Designed to compae 3S to other instrumentation tools.
- **Profile:** Records the ticks spent in each block.
- **Regex:** Records the program trace and compresses loops using a regular expression syntax.
- **TicksBoundary:** Records the number of ticks spent within each FPGA section. No additional instrumentation takes place to reduce the overhead.

Following is the description of the various 3S instruction level instrumentation tools:

- **Memory:** Records the memory access and in which block they occurred.
- **DynamicInstr:** Simply counts the number of instructions executed.
- **MemBounadryAcc:** Records the amound of memory that is required to be transferred to/from the FPGA section.

Chapter 2 Experiments

2.1 Platform

The experiments were conducted on the following platforms:

1. Intel Pentium M Processor with 1.8 GHz Clock Frequency, 512Kb L2 Cache and 1GB RAM

2. Dual Core AMD Opteron with 2.2 GHz Clock Frequency, 1MB L2 Cache for each core and 2GB RAM

Changing the execution platform did not have any significant effect on the results of our study because of the following reasons:

1. We use '-mpentium' gcc option during compilation because the 3S framework does not support the latest SSE extensions for Pentium M processor as well as the AMD Opteron

2. The results are subjected to comparative analysis, and therefore, do not depend upon the platform employed

3. The hardware accelerator concept is aimed at speeding-up legacy code which, in theory, cannot take advantage of contemporary microprocessor features for high performance computing such as Larger Caches and Memories, support for Superscalar/Vector processing such as Intel SSE Extensions, Multiple Cores etc.

4. If you look at the formula we use to estimate the speed-up, you will notice that increasing the clock frequency has the effect of magnifying the hardware acceleration/deceleration.

2.2 Strategies for Core Extraction

We explored two different strategies for automated extraction of execution cores in order to compare the respective performance and try to develop understanding of the quantitative, heuristics and hopefully qualitative parameters which determine the relative performance of the extraction.

The first strategy looks at the program on the whole in trying to identify the suitable candidates for subsequent implementation and execution on FPGA. On the other hand, the second strategy follows the Amdahl's law by identifying the functions which account for most of the execution time and concentrating the maximum efforts for the extraction of execution cores on these functions.

2.2.1 The First Strategy

The first strategy comprises of the following steps

1. Generation of the hotspot report using 3S

The first step involve the generation of the hotspot report using the 3S instrumentation framework. The hotspot report contains the frequency of execution of the various program blocks. By looking at the hotspot report, we can identify the program blocks with the highest execution frequencies in the benchmark. For example in case of the swim benchmark, partial hotspot report is shown in Figure 2.1.

```
Block            Order     ent       inst      ticks     tpe       tpi       line number

calc1_.L103.0.0  244       2621440   0         117233694 44        117233694 272
calc1_.L102.0.0  245       2621440   58        307783510 117       5306612   266,267,268,270,272
```

Figure-2.1 Partial hotspot report of the swim benchmark

Where:

- **block** is the block identifier
- **order** is the numbering in program order
- **ent** is the number of times the block is entered
- **inst** s the number of instructions in the block
- **ticks** is the number of ticks the block took to execute
- **tpe** is the ticks per execution
- **tpi** is the ticks per instructions
- **line number** is the line numbers of the program block in the original C/Fortran code.

In this case it can be observed that the block titled calc1_.L103.0.0 and calc1_.L102.0.0 with the program order 244 and 245 respectively are entered 2621440 times during the execution of the swim benchmark. Generally, it can be safely assumed that these program blocks manifest themselves in the form of loops. Similarly there are other loops which have a major contribution in the computational activity of the swim benchmark by virtue of a large number of iterations. By looking at the hotspot report, these loops can be identified easily.

2. Estimation of Speed-up

After identification, the loops which account for major contribution in the computational activity of the dominant function are enclosed within BEGIN_FPGA_N and END_FPGA_N markers, where N is an integer with the value 1, 2 or 3. These markers tell the 3S analysis tool that these loops have been identified as the execution core and they will be implemented in the FPGA to try and improve the performance of the program. We have limited the number of execution cores to 3 for each CFP95 benchmark. Also, we have assumed streaming communication for the estimation of speed-ups.

2.2.2 The Second Strategy

The second strategy comprises of the following steps:

1. Generation of profiling information for the benchmarks:

This step is performed using gprof, which is a popular profiling tool for gnu linux variants. gprof generates a quantitative execution profile of all the functions in the program. The functions which account for most of the program execution time are identified as a result. Latter steps employ this information to concentrate their efforts for the extraction of execution core on these functions. Figure 2.2 shows the gprof output of swim benchmark.

```
Flat profile:

Each sample counts as 0.01 seconds.
 %   cumulative   self              self     total
time   seconds   seconds    calls   s/call   s/call  name
33.14      0.54      0.54       10     0.05     0.05  calc1_
25.78      0.96      0.42       10     0.04     0.04  calc2_
24.55      1.36      0.40        8     0.05     0.05  calc3_
12.27      1.56      0.20        1     0.20     0.20  inital_
 2.45      1.60      0.04        1     0.04     0.04  calc3z_
 1.84      1.63      0.03        1     0.03     1.63  MAIN__
 0.00      1.63      0.00        1     0.00     0.00  global MAIN___GCOV
```

Figure-2.2 gprof output for the swim benchmark

Where:

- **% Time** is the percentage of the total running time of the program used by this function.
- **Cumulative Seconds** is a running sum of the number of seconds accounted for by this function and those listed above it.
- **Self Seconds** is the number of seconds accounted for by this function alone.
- **Self s/call** is the average number of milliseconds spent in this function per call, if this function is profiled, else blank.
- **Total s/call** is the average number of milliseconds spent in this function and its descendent per call, if this function is profiled, else blank.
- **Name** is the name of the function.

In this case, it can be observed that the execution core of swim benchmark comprises of three functions named calc1, calc2 and calc3. These functions account for 33, 26 and 25 percent of the total program execution time respectively. Subsequently, the latter steps of this strategy will attempt to extract the maximum constituent of the core from these three functions. This follows from the Amdahl's law as we are trying to accelerate portions of the code which are the slowest and hence hoping to get the maximum performance benefits out of our efforts.
Figure 2.3 illustrates the gprof outputs of all the benchmarks in the CFP95 suite.

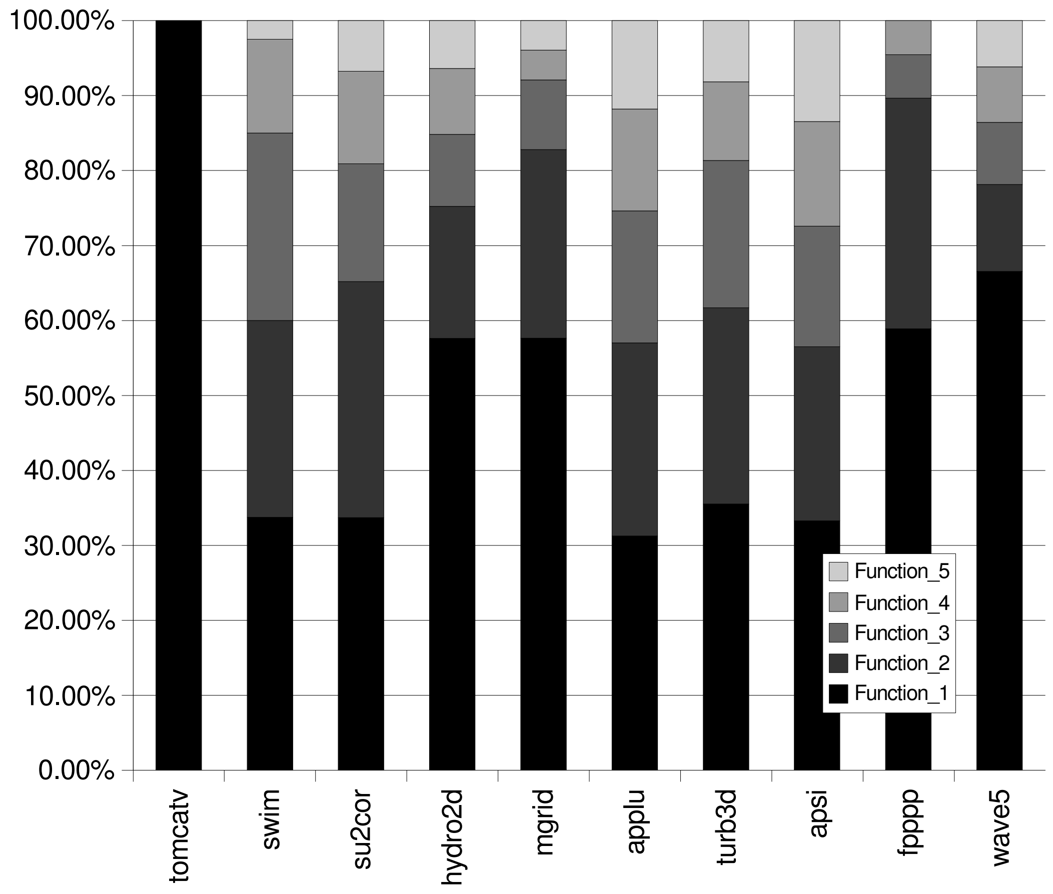


Figure-2.3 Percentage distribution of execution times for five most dominant functions in the respective CFP95 benchmarks. An uneven distribution is desired because an uneven distribution shows that the execution of the program is concentrated in one or two functions.

Following are some important observations regarding this figure:

- The most dominant function in each of the CFP95 benchmarks accounts for an average of approximately 45 percent of the total program execution time.
- The maximum percentage execution time for the dominant function in each of the CFP95 benchmarks is 100 percent in the case of tomcatv.
- The minimum percentage execution time for the dominant function in each of the CFP95 benchmarks is 14 percent in the case of apsi.
- The second most dominant function in each of the CFP95 benchmarks accounts for an average of approximately 19 percent of the total program execution time.
- The maximum percentage execution time for the second most dominant function in each of the CFP95 benchmarks is 31 percent in the case of fpppp.
- The minimum percentage execution time for the second most dominant function in each of the CFP95 benchmarks is 9 percent in the case of wave5.

2. Coverage analysis of the dominant functions

After identifying the dominant functions in our program, we run the gcov tool on the function using the source code annotation option to generate information about the execution frequencies of the respective program blocks within the dominant function. Generally, these program blocks manifest themselves in the form of loops. The loops with the maximum number of iterations are responsible for most of the computational activity inside the dominant functions. For example, in case of the swim benchmark, partial output of the gcov tool with the annotation options turned on is shown in Figure 2.4.

```
      10:  264:      DO 100 J=1,N
    10:  264-block  0
    10:  264-block  1
   5120:  265:      DO 100 I=1,M
   5120:  265-block  0
   5120:  265-block  1
2621440:  266:      CU(I+1,J) = .5*(P(I+1,J)+P(I,J))*U(I+1,J)
2621440:  267:      CV(I,J+1) = .5*(P(I,J+1)+P(I,J))*V(I,J+1)
2621440:  268:      Z(I+1,J+1) = (FSDX*(V(I+1,J+1)-V(I,J+1))-FSDY*(U(I+1,J+1)
    -:  269:     1          -U(I+1,J)))/(P(I,J)+P(I+1,J)+P(I+1,J+1)+P(I,J+1))
2621440:  270:      H(I,J) = P(I,J)+.25*(U(I+1,J)*U(I+1,J)+U(I,J)*U(I,J)
    -:  271:     1              +V(I,J+1)*V(I,J+1)+V(I,J)*V(I,J))
2621440:  272:  100 CONTINUE
```

Figure-2.4 gcov output for the swim benchmark

Figure 2.4 shows a nested loop in the calc1 function of the swim benchmark. The program block in the inner most loop is executed 2621440 times during the execution of the swim benchmark. Similarly there are other loops which have a major contribution in the computational activity of the calc1 function by virtue of a large number of iterations. By performing the gcov analysis, these loops can be identified easily.

3. Estimation of Speed-up

After identification, the loops which account for major contribution in the computational activity of the dominant function are enclosed within BEGIN_FPGA_N and END_FPGA_N markers, where N is an integer with the value 1, 2 or 3. These markers tell the 3S analysis tool that these loops have been identified as the execution core and they will be implemented in the FPGA to try and improve the performance of the program. We have limited the number of execution cores to 3 for each CFP95 benchmark. Also, we have assumed streaming communication for the estimation of speed-ups.

2.4 Speed-up estimation

2.4.1 Assumptions

Following the extraction of execution core of the program and accumulation of performance data based upon this extraction, we attempt to estimate the speed-up. In doing so, we make the following assumptions:

1. **Pipelining of Functional Components:** We assume that it is possible to chain the functional components of the program core so that they form an ideal pipeline. We neglect the delays due to filling and draining of this pipeline at the start and end of a computation respectively.

2. **No Structural Hazards in the Pipeline:** Given sufficient resources and the fine-grained parallelism of the FPGA architecture, we can assume that there will not be any structural hazards in the pipeline.

3. **Bandwidth Limited computation:** Now it is safe to assume that given an input vector, computation of the result vector can be accomplished approximately within a single clock cycle. This means that the computations are completely overlapped by the communication of input vectors to the FPGA and the result vectors from the FPGA.

2.4.2 Formulae for Speed-up estimation

Streamed transfer time is calculated using the following expression:

$$S \times max \frac{(memToFPGA, memFromFPGA)}{busBandWidth}$$

Non-Streamed transfer time is calculated using the following expression:

$$\frac{(memToFPGA + memFromFPGA)}{busBandWidth}$$

FPGA Optimised execution time is calculated using the following expression:

(Non-instrumented execution time) - (Time spent in FPGA section)
+ (Time spent transferring data across bus to FPGA)

Speed up is calculated using the following expression:

$$\frac{(Time\ of\ non-instrumented\ executable)}{(Time\ of\ FPGA\ Optimised\ executable)}$$

Where:

- **S** is the streaming constant with a typical value of 1.1
- **memToFPGA** is the total memory transfer to FPGA
- **memFromFPGA** is the total memory transfer from FPGA
- **busBandWidth** is the maximum data transfer rate for the bus which connects the FPGA to the microprocessor

  1.0 GB/s for the PCIE x4 bus
  2.0 GB/s for the PCIE x8 bus
  4.0 GB/s for the PCIE x16 bus

- Time spent in FPGA section is calculated by the following expression:

(Number of ticks in the FPGA section) / cpuFreq

Chapter 3 Results

3.1 Generic benchmarks results

3.1.1 Percentage distribution of Instruction types

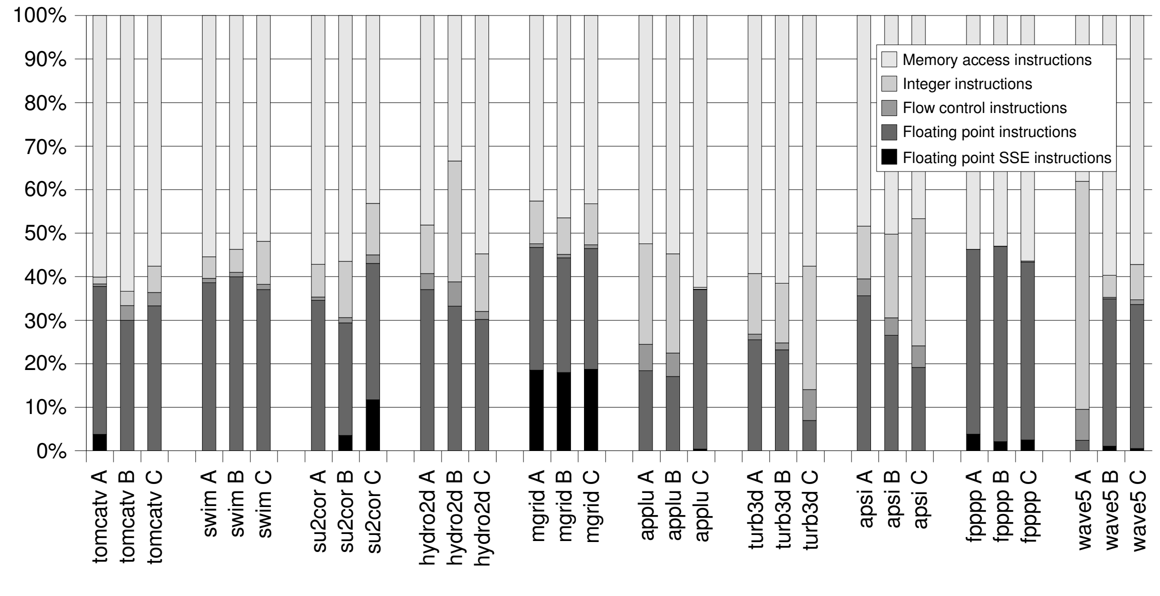


Figure-3.1 Percentage distribution of instruction types in the execution cores extracted using the first core extraction strategy

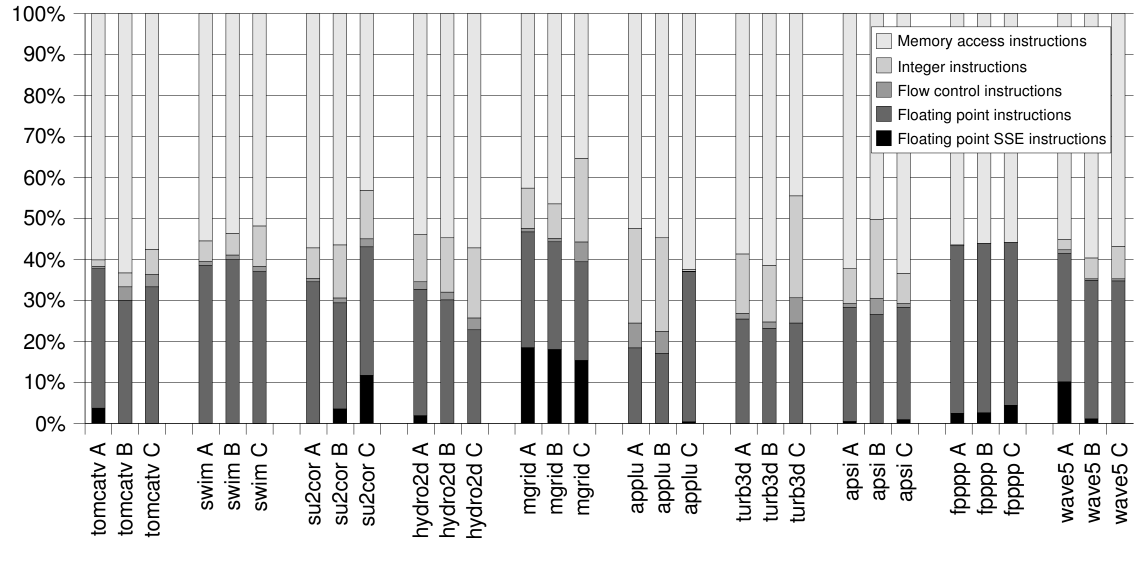


Figure-3.2 Percentage distribution of instruction types in the execution cores extracted using the second core extraction strategy

3.1.2 Memory Access

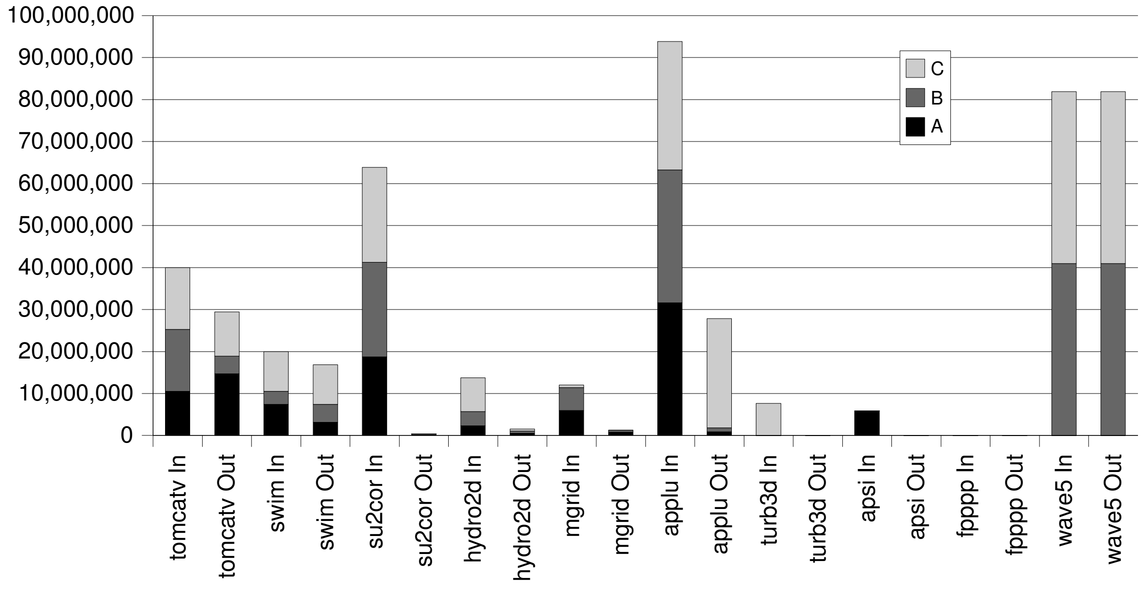


Figure-3.3 Memory access for the first core extraction strategy. A, B and C are memory access for the three execution cores extracted for respective benchmarks.

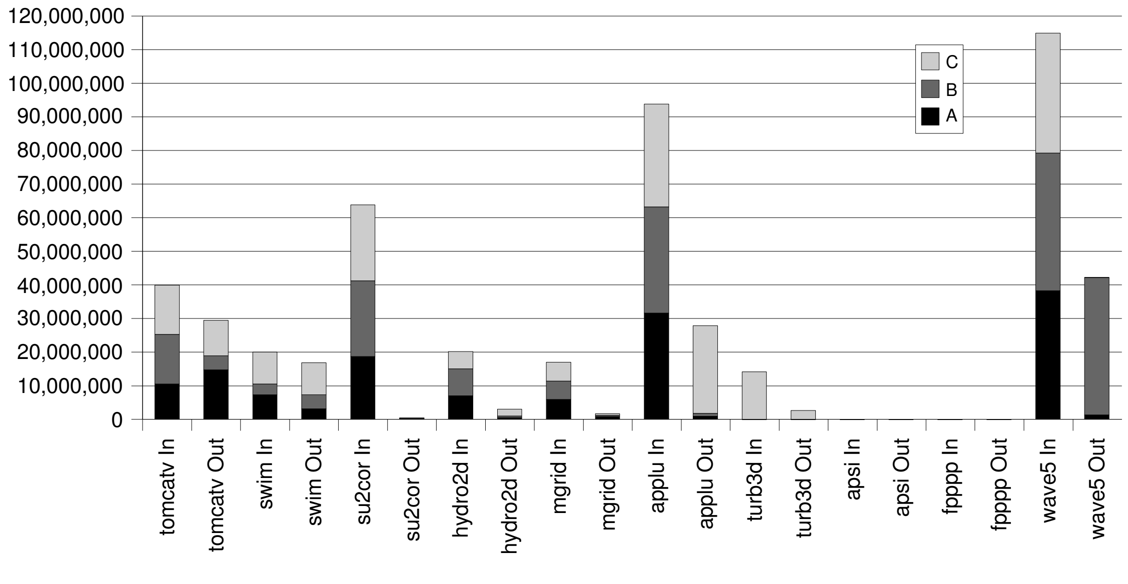


Figure-3.4 Memory access for the second core extraction strategy. A, B and C are memory access for the three execution cores extracted for respective benchmarks.

3.1.3 Percentage distribution of non-accelerated execution times

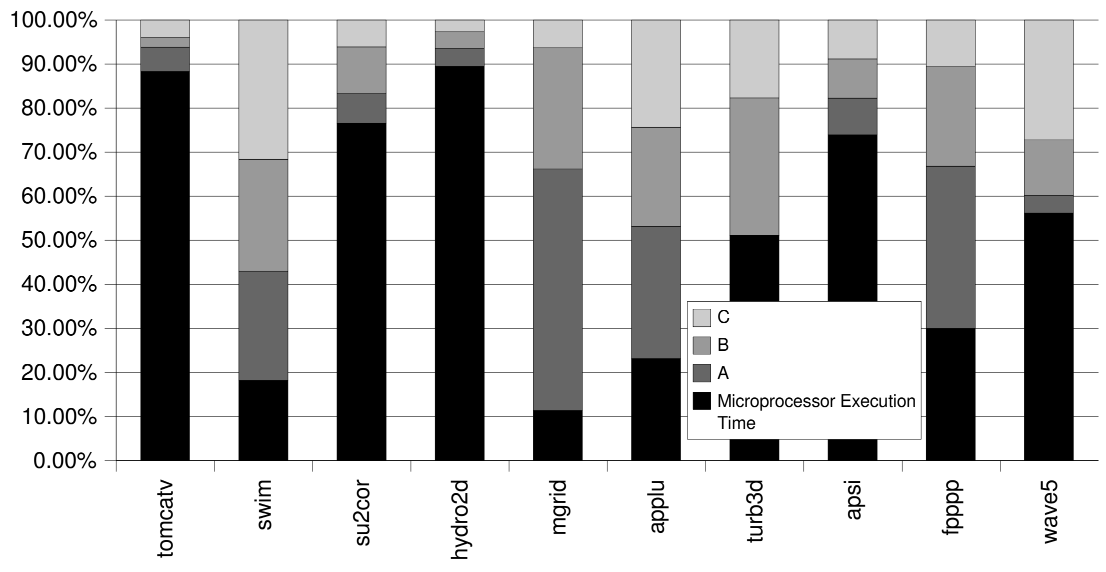


Figure-3.5 Percentage Distribution of non-accelerated execution time for the first strategy. A, B and C are the execution times for the three cores extracted for respective benchmarks.

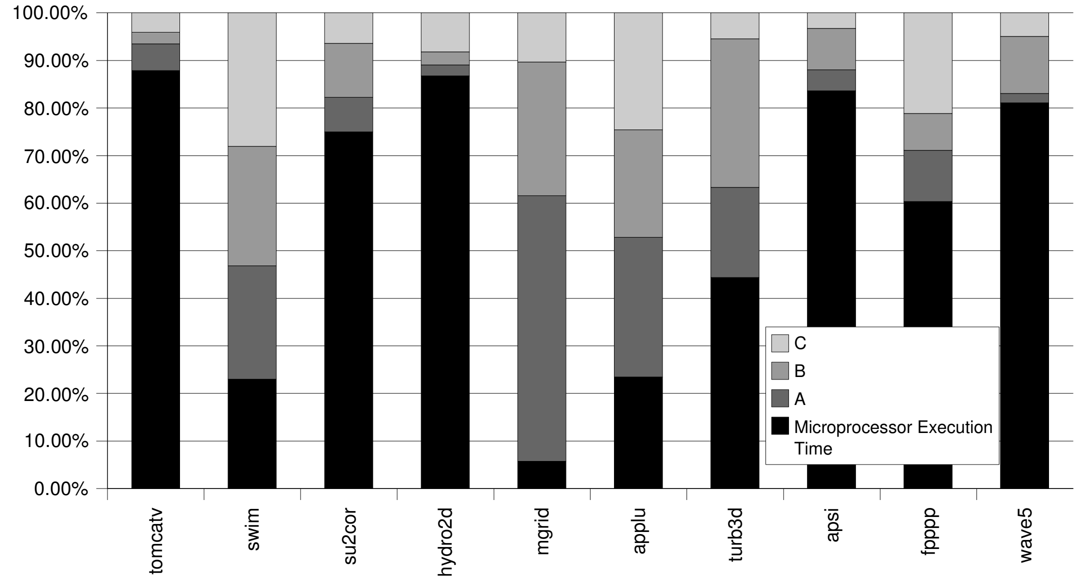


Figure-3.6 Percentage Distribution of non-accelerated execution time for the second strategy. A, B and C are the execution times for the three cores extracted for respective benchmarks.

3.2 Speed-up results

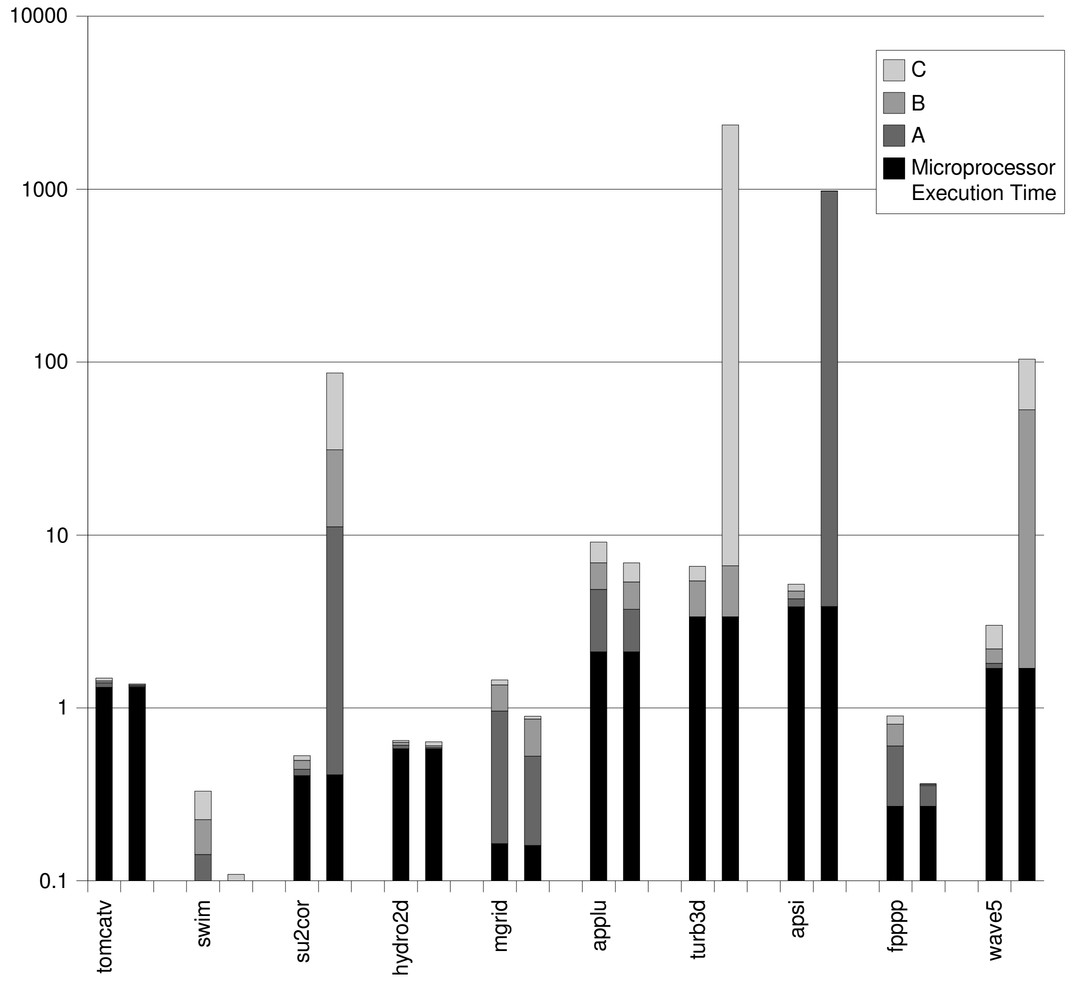


Figure-3.7 Comparison of non-accelerated and accelerated execution times for the first strategy. A, B and C are the execution times for the three execution cores for the respective benchmarks.

Following are the important observations regarding this figure:

- 21 out of the 30 execution cores (3 cores for every CFP95 benchmark) have shown speed-up
- The maximum speed-up estimate obtained for the CFP95 suite is by a factor of 1.42 for core 'A' in the mgrid benchmark
- The maximum slowdown estimate obtained for the CFP95 suite is by a factor of 357 for core 'C' in the turb3d benchmark
- The overall maximum speed-up obtained for the CFP95 suite is by a factor of 2.13 for the swim benchmark
- The overall maximum slowdown obtained for the CFP95 suite is by a factor of 417 for the su2cor benchmark

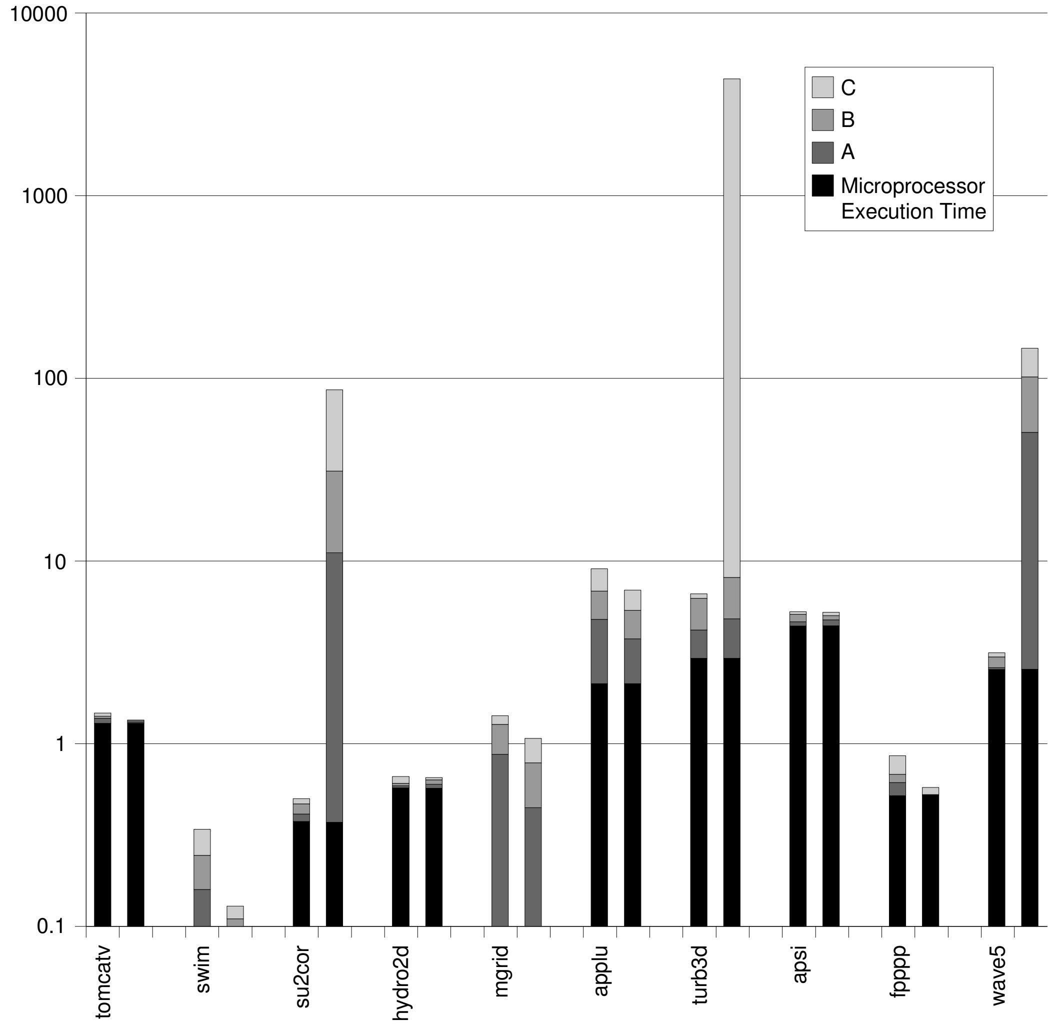


Figure-3.8 Comparison of non-accelerated and accelerated execution times for the second strategy. A, B and C are the execution times for the three execution cores for the respective benchmarks.

Following are the important observations regarding this figure:

- 16 out of the 30 execution cores (3 cores for every CFP95 benchmark) show speed-up
- The maximum speed-up estimate obtained for the CFP95 suite is by a factor of 1.5 for core 'A' in the mgrid benchmark
- The maximum slowdown estimate obtained for the CFP95 suite is by a factor of 667 for core 'C' in the turb3d benchmark
- The overall maximum speed-up obtained for the CFP95 suite is by a factor of 2.03 for the swim benchmark
- The overall maximum slowdown obtained for the CFP95 suite is by a factor of 454 for the su2cor benchmark

We can observe from figure 3.7 and figure 3.8 that neither core extraction strategy is able to achieve speed-ups for all the cores. There are 9 cores in case of the first strategy and 14 cores in case of second strategy which show slow-downs when implemented on hardware.

In a practical core extraction system, we will execute the core on the platform where it runs faster. Figure 3.9 and 3.10 shows the comparison of non-accelerated and accelerated execution times when all the cores are executed on the optimal platform.

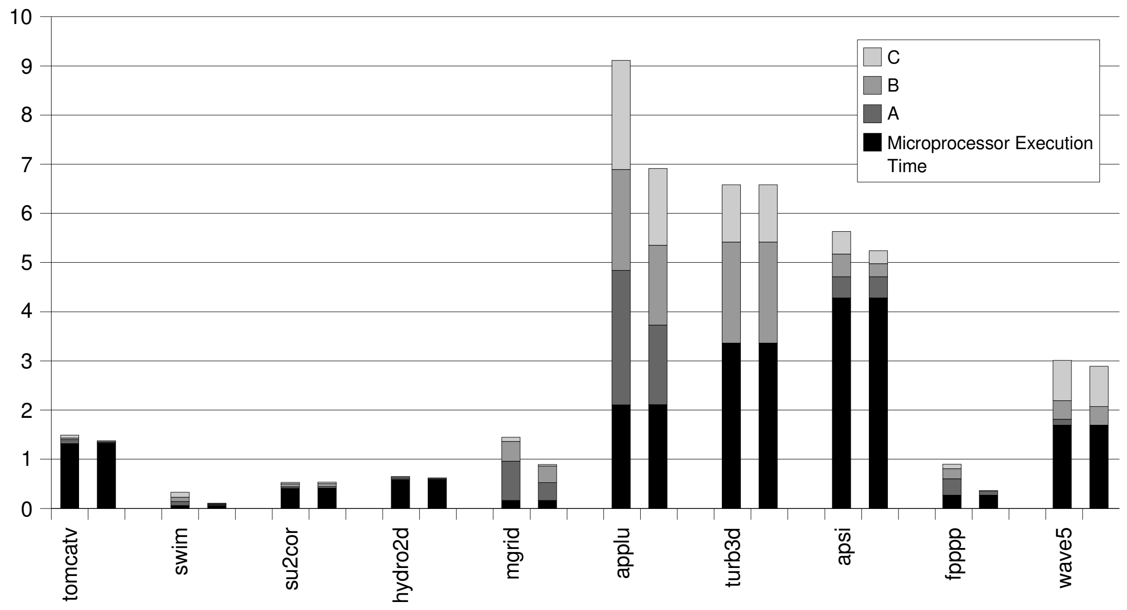


Figure-3.9 Comparison of non-accelerated and best accelerated execution times for the first strategy. A, B and C are the execution times for the execution cores for the respective benchmarks.

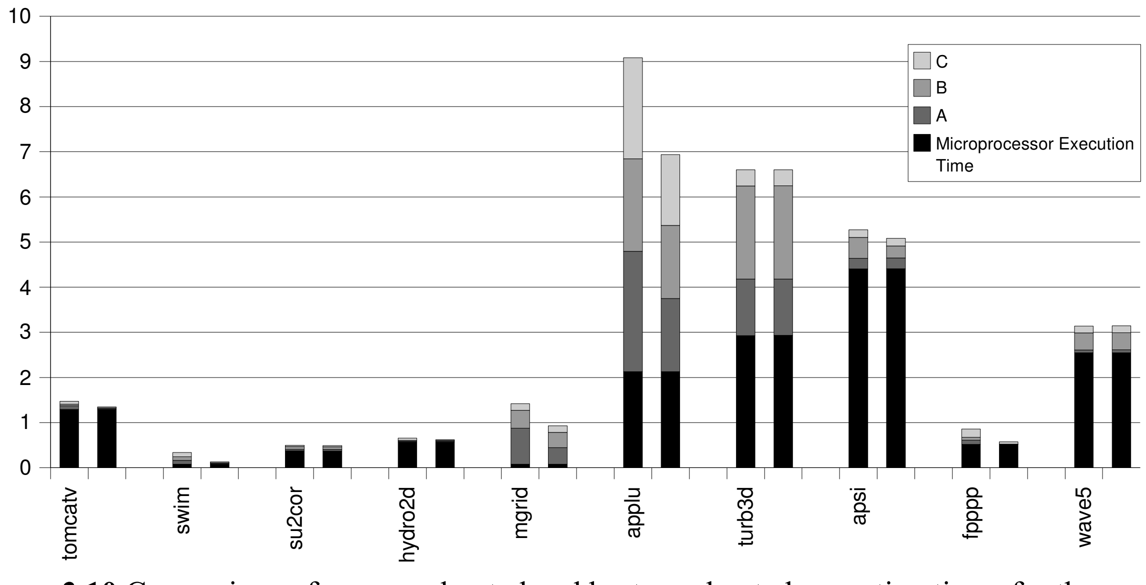


Figure-3.10 Comparison of non-accelerated and best accelerated execution times for the second strategy. A, B and C are the execution times for the execution cores for the respective benchmarks.

Chapter 4 Conclusion

We have studied the SPEC benchmark suite and attempted to identify and thus extract the respective execution cores using two different strategies. The first strategy looks at the program on the whole in trying to identify the suitable candidates for subsequent implementation and execution on FPGA. On the other hand, the second strategy follows the Amdahl's law by identifying the functions which account for most of the execution time and concentrating the maximum efforts for the extraction of execution cores on these functions. Subsequent to the extraction identification of execution cores, we estimated the potential speed-up obtained by implementing these cores on a reconfigurable device and examined the various parameters which affect the speed-up figure. Finally, we compared the results from the two extraction strategies. Following are some of the important observations regarding these results:

- Maximum overall speed-up for both strategies are estimated for swim
- Maximum difference between overall speed-up obtained by the first and the second strategy is observed for apsi and wave5 benchmarks
- In case of apsi benchmark,the second strategy performed better
- In case of wave5 benchmark, the first strategy performed better
- Maximum estimated speed-up was for core 'A' of mgrid benchmark in both strategies
- 19 out of the 30 execution cores extracted for each partitioning strategy are the same
- Out of these 19 execution cores, 10 show speed-up
- 11 out of the 30 execution cores extracted for each partitioning strategy are different

Applying loop transformations such as unrolling, fusion and fission were not included in the scope of this study. However, during the study we observed that applying loops transformations can result in higher speed-ups for many benchmarks and therefore is a good direction for future work. Of course, we can always achieve the highest speed-ups by transforming the algorithm so that it is optimal for FPGA implementation. However, such transformations are quite complicated and it is not possible or feasible to automate them.