ENTRA

318337

Whole-Systems ENergy TRAnsparency

Energy Optimization: Basic Static Techniques

Deliverable number:	D4.1
Work package:	Optimization (WP4)
Delivery date:	1 July 2014 (21 months)
Actual date:	22 August 2014
Nature:	Report
Dissemination level:	PU
Lead beneficiary:	Roskilde University
Partners contributed:	Roskilde University, University of Bristol, IMDEA Software Institute, XMOS Limited

Project funded by the European Union under the Seventh Framework Programme, FP7-ICT-2011-8 FET Proactive call.

Short description:
This deliverable describes the state of the art in the relevant energy optimization techniques together with a discussion of the optimizations to be applied to each case study.

The deliverable includes the following two attachments.

- D4.1.1. *Study of Possible Static Power Reduction due to Temperature Hot Spot Reduction provided by Uniform Register Utilization*. Technical Report.
- D4.1.2. *From Relational Verification to SIMD Loop Synthesis*. Published in the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13.

Contents

1 Summary

1.1 Static Optimization Techniques

This report is the first deliverable from work package WP4 (Optimization). The aim of this report is to identify and classify energy optimizations for code at various levels and the supporting analysis techniques needed to realize them. Tools for *energy transparency* enable a software developer to see how much energy an application uses and how the energy usage is distributed among the parts of the code. Work packages WP2 and WP3 focus on the fundamental modelling and analysis techniques needed to achieve energy transparency (see Deliverable D2.1 [EG13] and Deliverable D3.1 [LG13]). Given this information, the developer aims to apply optimizations to make the application more energy-efficient.

The report concerns only static optimization techniques, namely optimizations that are made by transforming the code or by fixing hardware parameters before execution. Dynamic optimizations, based on such things as run-time scheduling and dynamic modification of hardware parameters will be considered in deliverable D4.2.

A variety of tool support can be applied to achieve the required optimizations. In this report we consider optimizations independently of whether they are achieved by hand or with tool support, and we do not consider tool support in detail. The ENTRA project will develop prototype optimization tools, but we also note that energy transparency enables the developer to focus hand optimization efforts in the most effective way even if optimization tools are not available. Furthermore energy transparency assists the developer to evaluate the effectiveness of optimization, whether carried out by hand or with tool support.

1.2 Application to the project Case Studies

The survey of techniques considers a wide variety of hardware platforms, many of them beyond the scope of the current project. However, in Section 4 we perform a first assessment of the opportunities for optimization in the project case studies (work package WP6, deliverable D6.1). Our approach is as follows: we first outline the main architectural features of the target hardware platform for the ENTRA case studies. Secondly we summarize the context in which energy optimizations are performed, taking into account economic costs of development and manufacture. Following this we make a first assessment of the opportunities for optimization in the three case studies described in Deliverable D6.1 [Mul13]: real-time audio processing, robot and motor control and real-time networking.

2 Introduction

Given that software controls the way that the hardware consumes energy, it is clear that in order to minimize the total consumed energy of an application we need to design the software with energy consumption in mind. An estimate by Intel states that energy savings by a factor of 3 to 5 could be achieved using software optimizations alone [Edw11]. Correctly fitting software algorithms to the capabilities of the underlying hardware has been identified as the most important step in software design for low power, above all other power optimizations [RJ97]. The importance of exploiting parallelism is also identified, and has become of increasing significance as parallelization has become the dominant method of delivering higher performance.

To identify effective optimizations, all levels of the software stack need to be considered. Starting from algorithm and/or application specification, through the compiler which produces the low level code, down to operating system and machine code which controls hardware operation, these should be optimized in a way to minimize the total energy consumed. In this report we will discuss each of the mentioned levels in more detail.

The underlying hardware system also has to be taken into account, given that the optimization requirements are not always the same. For example, in battery powered embedded systems it might be important to reduce the *total* energy, and also the *peak power* since both have an effect on battery life. However, in large-scale general purpose systems the *average* computation power determines the amount of generated heat, and along with it the activity of cooling system, whose energy consumption can become significant. Thus, before starting the optimization, the optimizing problem and its objectives have to be clearly specified. We also consider various metrics for energy measurement [Mul13], covering various aspects of static and dynamic power (wattage) and total energy consumption. In the rest of the text, power and energy optimization will be treated as the same objective, given their close connection.

3 Classification of Optimization Techniques

3.1 Low-level Code Optimizations for Low Power

Research on power-aware compiler optimizations is not very extensive, mostly due to the lack of reliable and effective evaluation methods. There are methods that have been evaluated using physical measurements on a set of benchmarks, and also simulation based methods, but in general they were not considered to be sufficient proof of their practical utility. Thus, energy transparency is important in providing ways to evaluate optimization effects.

In this section we will first explain the difference between the compiler optimizations for

performance and the compiler optimizations for power and energy. After that we will present the most representative techniques divided into the following four groups:

- Thermal-aware compilation
- Register-file Energy Reduction
- Energy Efficient Compiler-based Task Parallelization
- Other important techniques that do not fall into any of the above groups

Finally, we will give a summary of the static analyses used in the cited techniques and draw the most important conclusions.

3.1.1 Power/Energy vs. Performance Optimizations

It has been widely accepted that existing compiler optimizations for achieving higher performance also achieve lower energy consumption given that higher execution time usually means higher energy. This is true for some of them, such as dead code elimination, common subexpression elimination, and in general all those that decrease the amount of work to be performed.

In these techniques we can also include all the loop optimization techniques that exploit memory hierarchy [RJ97]. The idea of these techniques is to introduce more locality in data accesses in order to be able to use the data stored in the parts of memory hierarchy closer to the processor, i.e., register file and/or cache. In essence, accessing the parts of the memory which are smaller and closer to the chip reduces the effective switching capacitance, thus reducing the energy consumption. However, some of these techniques introduce additional computation, thus increasing processor energy. For example, some linear loop transformations can result in complex loop bounds and array subscript expressions. Similarly, loop tiling can impose extra branch control operations. On the other hand, loop unrolling is expected to decrease energy consumption, since it decreases the impact of the loop overhead and in this way the total number of instructions.

All of this has been confirmed in the work of Kandemir et al. [KVI02], which explores the effect of the above-mentioned loop optimization techniques on energy. In particular, they evaluate energy consumption of matrix multiplication code on different cache configurations with different loop optimization techniques. Their conclusion is that the effectiveness of each technique depends on cache configuration, but in general the decrease of energy consumption does not follow the decrease of cache misses due to the effect of introducing additional computation in the core. Thus, they advise the use of these techniques along with other optimizations in order

to achieve savings on global level. In this way, they were able to reduce energy consumption by 55%.

One possibility presented in the same work [KVI02] is to combine loop optimizations, in particular loop fission, with the possibilities offered by hardware, in this case the existence of different memory banks that have different power modes. Loop fission in essence transforms a nested loop that contains multiple statements into a set of nested loops, each containing a subset of the original statements. In this way, the amount of memory needed in each loop iteration is reduced, which provides the possibility to use lower number of memory banks, and put the ones that are not used in hibernation mode.

In a similar fashion, compilers can help the operating system to decide when to turn on a particular power saving mode by inferring the time during which particular modules are inactive, so they can be turned off. The main issue with turning off modules is that their re-activation takes a lot of time. However, if the operating system knows when a particular module is to be used, it can start its activation and deactivation in a timely manner. An example of this is presented in [HPH+02], where an average of 70% reduction of disk energy is achieved. Another example is given in [SCO+07], where the compiler is able to derive disk access patterns, provided that it is aware of the disk layout. Based on the compiler-predicted future idle and active periods of parallel disks, a proactive disk power management can be implemented. In order to achieve additional savings, this information can be used for code restructuring in a way such that the length of idle disk periods is increased, which leads to better exploitation of power-saving capabilities.

Something similar can be done for Dynamic Voltage and Frequency Scaling (DVFS). For example, the compiler can infer the parts of the code where the processor can be slowed down with negligible performance loss [HK03]. Another example is presented in [SKL01], where the compiler provides an estimation of the execution time of each block. Hence, the static analysis necessary in this case is timing analysis. Here it is assumed that the worst execution time is the time of the critical path. Thus, if a block that is supposed to be executed next does not belong to the critical path, the operating system can calculate the slack time, i.e., the time difference between the block to be executed and the corresponding one that belongs to the critical path, and change the voltage and/or frequency in such a way that the execution time of the current block is (at most) the same as the execution time of the critical one.

A fundamental difference between performance and power optimizations is the model and/or metrics used. In the case of performance, the optimizations of the critical path are usually considered, while the optimizations that do not belong to the critical path usually do not affect performance. However, in the case of the power, each activity contributes to power consumption. A clear example of this is given by speculative activities, e.g., prefetching, which are executed based on a belief of a future behaviour. If the assumptions turn out to be false, additional work

has to be performed to undo the effects of speculative activities. If the critical path is not affected, it is clear that it will not affect performance, but obviously it can increase energy consumption. In the general case, this increase would have to be compensated by the overall energy benefit in order to be effective for energy optimization.

3.1.2 Thermal-Aware Compilation

In recent years a number of solutions for thermal aware compilation appeared aimed at reducing hot spots, i.e., the highest achievable temperature. In this way, it is possible to reduce static power consumption, given its super-linear temperature dependence and the fact that it is becoming an important part of power consumption, as well as energy spent on cooling in general purpose systems.

Researchers have identified the register file to be the hot spot of a processor, given its frequent usage and the way it is being accessed. Compilers usually assign a variable to the first free register, which means that the register located at the beginning of the register file will be used more frequently than the rest. For this reason, their temperature will be much higher than the temperature of the registers located further down the register file. In order to tackle this problem, the authors of the papers given in [AAB09, SAA10] propose to access the registers uniformly in order to avoid hot spot creation. This is usually performed by the compiler in an additional step called register re-assignment, using different techniques (heuristics mostly) to make the register usage more uniform. According to their results, hot spot temperature can be lowered by 11%. This can introduce significant reduction in cooling power in the case of general purpose systems. However, none of the papers give insight into static power reduction. We have performed some coarse grain estimations, according to which we could save up to 5% of static power with a register set of 12 registers in the cases of heavy register usage[1]. This means that these techniques cannot provide significant savings for embedded systems, and would be more appropriate for large scale general purpose systems.

A different approach is given in [MLN+06] by Mutyam et al., where the authors propose to identify thermal hot spots using software, as it is the software that determines the order and the frequency of accesses to different hardware components. The focus of the work is to provide compiler-based approach to make the thermal profile more balanced in the integer functional unit through load balancing. This can be in direct conflict with the idea of using the idleness of components to put them in a low power state, so appropriate load granularity should be found in order to reach the balance between the two. It is important to point out that this process is made without any performance loss, while the peak temperature is lowered by up to 14^oC.

[1]The calculations leading to these estimations are in Attachment D4.1.1

3.1.3 Register-file Energy Reduction

As mentioned above, the register file is accessed frequently, thus its energy consumption can become significant. For this reason, several techniques for reducing its energy consumption have been presented. However, they mostly rely on a specific hardware feature, which means that their applicability is limited.

One of the first ideas was to put in hibernation the registers that are not being used, given that applications typically use only a small part of register file [AVLV03]. This of course has to be supported by the hardware that should provide the possibility of turning off separate components. Similar work by the same authors seeks to reduce the number of read and write ports in a register file, reducing energy without significantly impacting performance [ALV05]. This is achieved through a combination of changes to the micro-architecture, and in the compiler, modifications to the register assignment strategy and the use of loop unrolling.

Another work [JOA+09] proposes optimal register caching in the architectures that support it. Register caching usually implies the necessity of adding extra logic to keep track of the information necessary for optimal caching, e.g., recent past or in-flight instructions, which means increased energy budget and higher chip price. Instead of adding extra logic, in this work it is proposed to keep available the information about complex data dependencies generated by the compiler, so it can be used for optimal register caching at run time. This information is added to unused bits in ISA. In this way, they achieve reduction of energy by 13%, at the same time increasing performance by 11%.

A subsequent work [GKD11] goes even further in proposing a register hierarchy. Different allocation algorithms are presented, all based on sharing temporary register files between concurrently running threads. In this way they were able to reduce register file energy by 54%. Having in mind that in GPUs register file can consume $15 - 20\%$ of dynamic energy, in this way they are able to achieve up to around 10% decrease of dynamic energy.

Another work tries to take advantage of so-called transport triggered architectures (TTA) [SHMC12], where the program has control over the datapath. In this way, it can reuse data stored in pipeline registers and in this way reduce the number of accesses to the register file. In this work the authors go even further by adding a backend to the compiler that performs operation-based instruction scheduling. The algorithm consists in finding the shortest path in a resource graph, where the weights of the edges correspond to the energy cost. In this way, they are able to reduce up to 80% of register file energy. However, the number of TTA-based processors is not big, which makes this approach very limited.

3.1.4 Energy Efficient Compiler-based Task and/or Data Parallelization

Nowadays having multiprocessors or multiple cores on the same chip is practically the standard, which provides the possibility of both task and data parallelization. Along with the possibility of voltage and frequency scaling, as well as turning off unused components, it can bring significant energy savings. Apart from multicore systems, parallelism is also supported in Very Long Instruction Word (VLIW) architectures through Instruction Level Parallelism (ILP), or in Digital Signal Processors (DSP) through ILP or Single Instruction Multiple Data (SIMD) instruction format. In the following we will present the role a compiler can take in this process.

One of the first works on this subject is presented by Azeemi [Aze06]. The author starts from the idea that performance and energy issues in embedded systems arise from inappropriate way software uses hardware. The work is tested on VLIW architectures, where up to five instructions can be executed at the same time. Thus, this work is an illustration of taking advantage of ILP. The compilation is implemented as iterative compilation, where the idea is to implement various versions of the same code using different optimization techniques, and select the best one. This process is implemented as multiobjective genetic algorithm, where the objectives are performance and energy, which are modeled using different monitors, such as code size, execution time, cache misses, etc. In order to maintaing the compilation time within reasonable bounds, search space is pruned using a heuristic. With this approach, they were able to achieve performance improvement of up to 80% for different benchmarks, with energy savings of up to 45%. However, all optimizations techniques are performance based, and it seems that the energy in this work is an additional benefit, rather than an objective.

In a similar way, the work of Lorenz et al. [LMD+04] represents a first attempt to generate SIMD instructions using a compiler through a vectorization step. This has allowed them to achieve on average 72% of energy reduction with average 76% performance improvement on a set of benchmark applications. Although nowadays it is a common thing for a DSP compiler to generate SIMD applications, this work is a good illustration of the benefits it provides for both performance and energy.

However, existing pattern-based compiler technology is unable to effectively exploit the full potential of SIMD architectures. In order to overcome this limitation, a new program synthesis based technique for auto-vectorizing performance critical innermost loops is presented in [BCG+13][2]. The synthesis technique is applicable to a wide range of loops, consistently produces performant SIMD code, and generates correctness proofs for the output code. The synthesis technique, which leverages existing work on relational verification methods, is a novel combination of deductive loop restructuring, synthesis condition generation and a new inductive

[2]Appears as a Attachment D4.1.2

synthesis algorithm for producing loop-free code fragments. The inductive synthesis algorithm wraps an optimized depth-first exploration of code sequences inside a CEGIS loop. The technique is able to quickly produce SIMD implementations (up to 9 instructions in 0.12 seconds) for a wide range of fundamental looping structures. The resulting SIMD implementations outperform the original loops by 2.0-3.7. Although the technique is mainly applied to execution time optimization, it can also reduce energy consumption.

The connection between task parallelization, voltage and frequency scaling and selective turning off of different components on a multicore chip is investigated in the work of Cho and Melhem [CM10], which derives fundamental formulas to describe the connection between parallelizing an application, its performance and energy consumption.

A concrete implementation of this approach is given by Chen et al. in [CDYW05] . The main idea of this approach is to use load imbalance of parallel applications, i.e., the fact they will not finish their part at the same time and some of them will have to wait for others to finish, and modify voltage and frequency of separate processors such that they all finish their task in (approximately) same time. Apart from parallelizing a serial application, the compiler has two more tasks: it should perform load imbalance analysis, i.e., estimate the load of each parallel and serial fragment, and in the second step calculate the voltage and the frequency of each core, so they all finish their task at the same time. However, since in many cases the load depends on input data, it cannot be estimated at compile time (in the best case its function on the input data can be inferred), which makes this approach inapplicable in these situations.

A continuation of the same idea is presented in [OKC11] by Ozturk et al. The idea of this work is to provide compiler support to multimedia embedded applications (characterized by a great number of nested loops) on the chips with voltage islands, i.e., different areas of the chip with their own variable power supply. Their work is also based on load imbalance. However, they go further by identifying the main source for this in multimedia applications to be loop bound based imbalance. This is the only type of load imbalance that could be estimated at compile time.

This work further exploits both data parallelism, i.e., performing a similar computation on different data, by executing a given loop nest in parallel on different cores, and task parallelism, by dividing a given loop nest into independent parts (if possible), and executing each of them on a different core. After mapping each nested loop and its parts on different cores, the second step is to estimate the load of each core. In order to do this, the compiler should perform two calculations:

1. Iteration count estimation

2. Per-iteration cost estimation

The first step is not very difficult, given that in most array-based applications loop bounds are known before the execution starts, or they can be estimated through profiling. The second step is more complicated, and in general case can be estimated using a Worst Case Execution Time (WCET) technique, which is well known. The approach used in this work is based on the number and the type of the assembly instructions used in each loop.

After having estimated load of each core, the voltage of each one is assigned in the following way. The one with the biggest load will have highest possible voltage, the second one will have the lowest possible voltage that does not increase the total execution time, etc. In this way, they were able to reduce energy consumption by 40.7% on average, achieving at the same time 14.6% average performance improvement in the terms of the total number of execution cycles.

3.1.5 Code Motion for Energy Optimization

Code motion means rescheduling or reordering execution of instructions, while preserving the functionality of the code. In many cases energy saving can be achieved by code motion. In the real world, programmers are seldom aware of the hardware energy-usage implications of a particular ordering at runtime, and this often leads to loss of energy-saving opportunities.

There are two basic scenarios in which code motion can play a key role in reducing energy consumption: in optimising time-critical code and in smoothing workload spikes. A section of code may be time-critical due to its role in inter-process communication, a real-time I/O response or other time-critical operation. If code in the critical session, which is irrelevant to the critical operation, into the non-time-critical session, can facilitate run-time energy optimization, as shown in the following example, relating to "user-perceived time" [SSCK14].

To explain user-perceived time, we take a look at one interactive session (e.g. a click on a button or slide on the screen). The user first feeds in an input, and the system will render a UI response after a period of processing. The interval between user input and UI response is the so-called "user-perceived time". In most cases, there is some "think time" between the UI response and the next user input. As far as the user is concerned the UI response should be as fast as possible (it is "time-critical") whereas the user is most likely oblivious to the processing during think time. Thus we can apply aggressive runtime optimization (e.g. slowing down processor frequency) during the think time. Further, we can delay code that is irrelevant to UI response (code motion) so that processor frequency can be reduced during both think time and user-perceived time, while still guaranteeing timely UI response.

The second basic scenario mentioned above concerns workload spikes. A heavy workload that comes in a short time slot causes workload spikes. Due to sluggish (coarse-grained) run-time power adaptation, the processor needs a high frequency level to deal with workload spikes;

however, keeping the same high level even between and after the spikes leads to a waste of energy. To reduce workload spikes, high power instructions should be spread over a long period. There are many challenges and trade-offs to be addressed, such as preserving the user experience.

Performing code motion is in general a complex task. In the simplest case (in one single code block), code motion consists of delaying instructions that are irrelevant to the critical statement(s) (i.e. the statement responsible for time-critical operation) until after the critical statement, but respecting dependencies of other statements on the delayed instructions. That is, an instruction cannot be delayed until after another instruction that depends on it. It is tricky to find the relevant code for the critical statement, which is the same process as find the irrelevant code, and the process is formally called program slicing [Wei81]. A program slice consists of the parts of a program that (potentially) affect the values computed at some point of interest. The calculation of the slice is based on the Program Dependence Graph (PDG) [HRB88], which consists of the Control Dependence Graph (CDG) and the Data Dependence Graph (DDG). A statement is in the program slice for the interested statement if there is a path connecting them in the PDG.

In inter-block cases, there are many trade-offs. For example, moving code out of a loop may induce more spikes due to frequent processing of high power instructions without interspersing low power instructions around them in the loop. Accurate energy modelling is needed to evaluate the benefits of code motion in more complex cases.

3.1.6 Superoptimization

Traditional compilers optimize code during compilation, however, they only produce improvements over the existing code instead of truly optimal code. Superoptimization is a technique developed by Massalin [Mas87], allowing an optimal section of code (for a given metric) to be found for a given function. By searching through all possible instruction sequences and checking whether the target instruction sequence is equivalent, an optimal piece of code can be found.

Massalin found that by applying superoptimization sequences of code were found, which exploited the processor's features in unusual ways. These sequences of code made use of the flags, along with combinations of arithmetic and bitwise operations.

```
1  int signum(int x)
2  {
3    if(x > 0)
4      return 1;
5    else if(x < 0)
6      return -1;
7    else
8      return 0;
9  }
```

```
1  ; d0 = n
2  add.l  d0, d0
3  subx.l d1, d1
4  negx.l d0
5  addx.l d1, d1
6  ; d1 = signum(n)
```

The code above on the left shows one of the functions Massalin optimized, with the resulting Motorola assembly on the right. The resulting assembly makes use of the carry bit (`x`) to perform the computation. The carry bit is set by the first operation if the input is negative, and this bit is propagated into `d1` in the next instruction, which computes `d1 = d1 - d1 - x`. The third instruction negates `d0`, which sets the `x` bit if `d0 != 0`. Finally the values in `d1` and `x` are combined to give the result. The optimal version of this function was shorter than any previously found instruction sequence — an expert writing the same sequence managed to reduce it to 6 instructions.

In general superoptimizers have a similar structure:

1. Enumerate the instruction sequence.

2. Test the instruction sequence.

3. Cost the sequence.

When the entire set of instruction sequences has been enumerated, the instructions can be ranked and the lowest one selected. Often superoptimizers will attempt to enumerate the sequence in cost order, allowing the first equivalent sequence to be selected and the search stopped. In the case of optimizing for code size this is simple — the instruction sequences are generated in size order. This method becomes more difficult if targeting performance or energy consumption.

Some superoptimizers have attempted to reduce the size of the search space by canonicalizing register namings [BA06], by restricting the number of constants used [GK92] and by removing redundant instructions (such as X = Y, Y = X) [Mas87].

Testing the instruction sequence is a challenging problem, as it must be done fast, yet ensure that the instruction sequence is equivalent for all inputs. One approach taken by many superop-

timizers [BA06, SSA13] is to test the instruction sequence on a few test vectors, and if it passes all of them, verify the sequences are equivalent using an SMT solver.

Other approaches have attempted to guarantee the instruction sequence is correct by construction — this incorporates steps 1 and 2 together. Gulwani et al. [GJTV12] use a solver and counter example guided inductive synthesis to connect multiple 'components' together to produce similar types of programs to Massalin's superoptimizer. A similar approach is taken in [BCVF06] using answer set programming to produce a set of instruction sequences.

There are currently no superoptimizers which explicitly target energy consumption, although improvements in energy consumption may be expected from reducing code size, and increasing performance.

3.1.7 Other Important Techniques

Provided that the compiler has insight into energy consumption, other common compiler techniques could be used taking as objective energy rather than performance. For example, peephole optimization, which changes a piece of code with another that carries out the same task, can perform the change in a way the new code consumes less energy. In general, it is possible to have different versions of the same code, and at compile time decide which one to use. Also, if precise energy consumption cannot be estimated without the data, with the use of Just In Time (JIT) compilers it is possible to decide which is the most optimal version of the code (in terms of energy) at run time and continue its execution. However, this introduces certain time overhead at run time. In the following text we will see some examples of algorithm and/or code transformation that can provide energy savings.

In a similar way, instructions can be scheduled so as to reduce the energy, given that the state of the hardware, as a direct result of the previous instruction, also affects on its energy consumption. An example of this is so called *cold scheduling* [SyTD94], whose aim is to reduce interinstruction effects. Another example is given in [PKVI00], where Parikh et al. can save up to 17% of energy through optimal instruction scheduling.

Finally, the compiler should be aware of the hardware and the possibilities it provides. For example, if there is a floating point unit, the compiler should generate the code that uses it. Thus, the back-end should be designed, or optimized if possible, in such a way that it takes advantage of all hardware possibilities.

3.1.8 Summary of Used Static Analysis Techniques

We will now list all the static analyses used by the above-mentioned compilers as inputs to the optimization step, or identified as a necessity:

- Energy accounting (energy transparency), i.e., providing insight into the amount of energy a piece of code spends, in order to enable the evaluation of different optimization techniques
- Inferring the time (starting and ending point) the components (e.g., disks) are not active, which can be used by the OS to turn them on and off in a timely manner
- Identify parts of the code the processor can be slowed down with no performance loss, e.g., memory access, as an enabler for voltage and frequency scaling
- Load imbalance analysis, as a special case of the previous item
- Execution time estimation, as another enabler for voltage and frequency scaling
- If there is more than one resource for a certain action, find the shortest path to it (in the terms of energy)

3.1.9 Concluding Remarks: Compiler Optimization Techniques

In this section we have presented few representative compiler optimization techniques. The most important observation is the lack of energy accounting which would provide the compiler the necessary insight into the quality of each optimization technique without having to perform simulation and/or measurements. On the other hand, although there are lots of different examples, there is no general approach that would combine different techniques in order to obtain overall energy minimization. As we have seen, some techniques can have positive impact on the consumption of one part, but may increase the consumption of another. For this reason, it is important to treat the optimization problem on a global level. Finally, the compiler should be aware of the underlying hardware in order to be able to use all its potentials for saving energy.

3.2 Algorithm and Source-Code Energy-aware Optimizations

In this section we will present solutions aimed at optimizing energy through algorithm or high level code modifications. We have identified the following major trends among modifications:

- Re-computation vs. Communication
- Code level parallelization
- Trading precision, or Quality-of-Service (QoS), with energy

3.2.1 Re-computation vs. Communication

Technology scaling has been more beneficial to transistors than to wires. For this reason, communication has become the limiting factor of both power and performance [MG08], especially bearing in mind its growing significance in today's multi-core era. Even the introduction of the Network-on-Chip (NoC) paradigm [BWM+09] does not solve the problem completely, given the growing trend of communication requirements.

A solution to this problem is to perform re-computation of some information, rather than fetching it from a remote place [MG08]. An important enabler for this approach is to develop greater understanding of algorithms and data structures in order to better manage data movement in systems. Furthermore, it is important to be able to estimate the cost of both computation and communication in order to be able to decide which one is more beneficial in particular cases. The authors believe that the significant work done in VLSI domain in characterizing and predicting interconnections can be helpful in understanding communications in multi-core processors.

The great majority of publications on the subject have the objective of increasing performance by trading re-computation for communication. Although without knowing their energy cost we cannot claim that these techniques provide energy savings as well, they can be a good starting point.

One example is given in [RK08], where Rolf and Kuchcinski give a solution to a parallel depth first search algorithm, where they show how reducing communication at the expense of increasing computation can achieve speed-up of up to 18 times. They also claim that less communication gives more room for an advanced load balancing scheme, which can further improve performance. As we have seen above, load balancing can also be used to reduce energy.

Another example consists in honouring multiple alignments, i.e., relative allocation within an array, when decomposing data into distributed memory [GSB95], which reduces conflicting alignments and in this way the communication. The performance increase is 80%, which becomes more significant bearing in mind that when it was published (1995) communication did not represent such a significant overhead.

3.2.2 Code Level Parallelization

As previously mentioned, executing code concurrently on multiple cores can be energy efficient [KA10, KA11, KA09a]. It is also possible to optimize performance through static energy-bounded scalability analysis [KA09b], given an energy budget. Similarly, devices capable of parallelism through hardware multi-threading may require some level of parallelism in order for a core's energy efficiency to be maximised [KE13, SB13].

Although the process of parallelization is often performed automatically by the compiler, parallel code can be developed manually by the programmer. This process can be time-consuming, complex, iterative and error prone, yet it is more flexible and it can be the only solution if the compiler is not capable of generating parallel code. An extensive survey of parallel programming models and tools can be found in [DMCN12].

3.2.3 Precision (QoS) - Energy Trade-off

In the recent past some researchers have studied energy-accuracy trade-offs [LY07]. The main conclusion is that there are applications in which a significant proportion of energy is spent on providing a *correct* result, whereas the applications are resilient to error [LY07, MSHR10]. One of the most important conclusion from these studies is that there are parts of applications that are resilient to errors, or provide a result that is good enough, while there are parts that need to have the precise result.

This idea is exploited in the design of EnerJ [SDF+11], an extension to Java which provides a solution for isolating precise parts of the program from the ones that can be approximated by introducing approximate types, i.e., type qualifiers that declare that the data can be used in approximate computations. The system can use this information as a sign to use low power storage, low power computation, or even low power algorithms provided by the programmer. Furthermore, the system can statically guarantee the isolation of precise program parts from the approximated ones, which eliminates the need for dynamic checking and the additional energy consumption implied with it. In this way, energy savings of 10% to 50% can be achieved.

However, in order to provide the practicality of the approach, it has to be supported by the hardware as well. The authors propose an ISA extension, where the additional bits would show if the components that support the execution of the approximate code should be involved in the current calculation. They also envision the existence of separate ALU and floating point units for both approximate and precise operations. Furthermore, the cache lines should have an additional bit for distinguishing the approximate from the precise ones. It is also proposed to supply the approximate memory (SRAM) with lower voltage, or refresh DRAM with lower frequency, etc. However, the additional hardware would lead to extra chip fabrication costs. Thus, in order to support the practical exploitation of EnerJ-like proposals, it should be evaluated when is the additional hardware cost justifiable.

An important contribution to this line of research has been provided by the Computer Science and Artificial Intelligence Laboratory from MIT [MRR11a, MRR11b, ZMKR12, MSHR10, RHMS10]. Their work started with the idea of supporting the execution of a task in the case of having to cope with limited resources, failures, etc. [RHMS10], and eventually it evolved to the

idea of providing optimal accuracy-resource consumption trade-offs [ZMKR12], where resource can be time, energy or cost. It is clear that in the case of energy an important enabler for this process is the estimate of energy consumption. Another important enabler is the awareness of which parts of the algorithm permit lower accuracy, i.e., in image processing pixel values can be approximated, but the part of the code that determines how to reconstruct the image has to be accurate.

The examples of accuracy-aware transformations they give are task skipping (in parallel applications) [Rin06], loop perforation, i.e., executing only a subset of the original loop iterations [MRR11b, Aga09, SDMHR11], approximate function memoization [CGL10], substitution of multiple alternate transformations [BC10], such as shifting left or right instead of performing division or multiplication, or sampling, which assumes discarding certain computations. In the case of reduced resource computing, we could also add cyclic memory allocation [NR07], where a fixed sized buffer is allocated for a given dynamic allocation site. These transformations provide the possibility to realize different optimization problems, such as to minimize resource consumption subject to a given accuracy, or to maximize accuracy subject to a given resource consumption. A solution to these optimization problems is given in [ZMKR12], while in [MRR11b, MRR11a] a probabilistic reasoning that justifies that the application of transformations may change the result within probabilistic accuracy bounds is given for the case of loop perforation.

The same research group also provides PowerDial [HSC+11], a system that dynamically adapts the application to the varying characteristics of the environment (in terms of load, power, resources, etc.). In essence, it is based on a set of static configuration parameters which can be transformed into *dynamic knobs*. The dynamic knobs can be further tuned by the control system to change the configuration of the running application in order to trade off accuracy of the computation in return for the computation resources the application needs. Although automatic, this technique requires the user to provide training data, an output abstraction and identify a set of parameters (static configuration parameters) and their range in the program.

3.2.4 Summary of Used Static Analysis Techniques

The enabler for all the deployed techniques is again the estimation of the consumed energy. Other static analysis techniques use to enable the code transformation are the following:

- Understanding communication patterns in order to enable communication - re-computation trade-off

- Static energy-bounded scalability analysis, which optimizes performance of parallel algo-

rithms given an energy bound

- Static verification of approximate and precise code computation, in order to enable EnerJ-like applications
- Probabilistic reasoning which justifies that the applications of code transformations change the result within given accuracy bounds, in order to enable approximate computations

3.2.5 Concluding Remarks: Source Code Energy Optimization

This section has provided some insight into code transformation techniques that can reduce energy consumption. Particularly interesting is the application of approximate computation which can be applied in many applications within acceptable losses. Although it has been mostly applied to performance optimizations, it could also be applied for energy optimizations, assuming that estimation of energy consumption is available.

4 Optimization in the Case Studies

4.1 Architecture and considerations

Before discussing the optimizations that are relevant to various case studies, we discuss architecture and system characteristics that may be exploitable for energy optimization, or conversely, that may constrain or preclude certain techniques. The classifications of optimizations described in Section 3 are referenced where appropriate.

Exploitation of available parallelism can be achievable at various levels, but the best to apply may be architecture and program specific. For example, VLIW DSP processors benefit from aggressive instruction scheduling. The instruction set and processor pipeline are designed to exploit instruction level parallelism, and the signal processing they perform fits well to this structure.

In the case of the nominal target for these case studies — the XS1-L — there is a reasonable amount of parallelism possible per core (up to eight threads), but no instruction level parallelism. However, there are combined arithmetic operations such as *multiply-accumulate*, which speed up common data processing operations. Further, for Input/Output, port resources can handle some data manipulation tasks, such as buffering, timing and serialization.

4.1.1 Thread parallelism

In [KE13] the energy efficiency of the XS1-L in relation to the number of utilised threads is discussed. If fewer than four threads are utilised per core, then the pipeline is under-used, resulting

in sub-optimal energy efficiency. As such, this motivates keeping the pipeline full whenever the processor is active, by having four or more active threads. This is true of other architectures with parallel features, such as the Xeon Phi [SB13], which has sub-optimal efficiency if fewer than two threads are running on a core.

The methods by which maximum pipeline efficiency can be obtained is governed by the dependencies and structure of the application. As such, for each case study, optimization strategies may differ.

In terms of the overhead of parallelizing in the XS1-L, at least three instructions are required to create a thread, and further instructions may be required in order to setup all registers such as stack, data and constant pointer. This constrains parallelization optimizations that would create new threads, to ensure that the work they will perform is sufficiently large compared to the thread creation time. Alternatively, provided there are sufficient thread resources available, the threads may be permanently allocated, and wait for work to be communicated to them.

Beyond maximising pipeline utilization, another possible outcome of optimizing for thread parallelism is that it may then be possible to lower the processor clock frequency and voltage. The application remains able to deliver the required performance (e.g. the same data throughput), whilst consuming less energy, a trade-off discussed in Section 3.1.1.

4.1.2 I/O and computations

Some architectures have been designed so that real-time I/O and communication can be scheduled relative to computation. One of our target architectures (the XS1-L processor) is a good example of this. The central idea is that when programming the system, the program is seen as a sequence of I/O operations and computation operations:

```
Perform some computation
Perform some I/O
Perform some computation
Perform some I/O
```

When an architecture is designed to dissociate I/O and computation, then the precise timing of the I/O is no longer an issue: the I/O instruction has to be reached *before* the required time; but not *at* the required time. This means that the computation has to be *fast enough* to perform the I/O on time, but it can be made to run faster without affecting the semantics of the system.

Architecturally, I/O and computation can be dissociated by triggering the I/O on a particular clock cycle (for example, the next clock, or the fifth clock), or by not allowing the I/O to proceed until some external event has completed, (for example the first clock where a handshake wire has

been asserted). Note that in many architectures this is not the case, and I/O has to be delayed to the appropriate time by inserting no-operation instructions (NOP) in the instruction stream.

Dissociation of I/O and computation is important from a power perspective as it enables both the programmer and the tools to schedule computation tasks around the I/O instructions; and the only constraint is that the I/O instructions are reached in time, and that any data dependencies are maintained. If these goals are met, then given a fixed throughput of the processor, it can be proven that the system will work. Conversely, given knowledge as to how much slack there is in the system, the clock frequency of the processor can be delayed so that the computation finished *just in time*; which in turn will enable us to lower the voltage, and make significant power savings. Having sequences of no-operations inserted take this freedom away, or put the responsibility completely in the lap of the software engineer.

A further optimization is to exploit parallelism between I/O and computation. If I/O and computations have been dissociated, then as a buffer between the two will enable computation to continue, whilst the I/O unit is ready to perform the I/O; synchronising on the next interaction, at which case the computation has to wait for the previous I/O operation to complete. This gives further opportunities for scheduling computations.

4.1.3 Core parallelism

Where multiple cores are used, voltages and frequencies can be optimised per core. Various of the techniques discussed in the previous section may be used or combined for optimising, where there is core-local parallelism, multi-core parallelism, and inter-core communication to consider. In particular, consideration should be given to re-computation vs. communication (Section 3.2.1, as well as task parallelization (Section 3.1.4).

4.2 Economics of power optimizations

As stated before, energy use can be optimised by using a multitude of cores that run slowly. Given the limits of voltage scaling, the physics of leakage, and the extra costs incurred in communication, an optimal number of cores can be estimated that will solve a problem using the least energy. However, from a commercial perspective this is not always desirable as deploying extra hardware costs money.

Indeed, when designing a system, the designer is often given budgets within which to stay. In the case of a mobile system, the budget will be for the for the maximum weight and size, the maximum bill-of-material cost (the BOM cost), and the minimum time that the system can operate on a single charge. In the case of a non-mobile system, the budget may be the energy

usage and bill-of-material costings (the BOM cost). In that case, the energy usage is often guided by legal requirements. This may be the required maximum stand-by power usage, or a desired efficiency label.

In either case, power does not need to be optimised beyond the point where it is "good enough". Reversely, it may be that parallel tasks have to be combined in order to reduce the system cost; the designer may have gone over the cost-budget, but be below the power-budget. All optimizations have to be balanced too; when all budgets are met, it may make sense to make system half the cost whilst decreasing power efficiency by 5%; or it may be worthwhile to increase cost by 5% in order to double power efficiency.

4.3 Real-time audio processing

A good case to address with the aforementioned techniques is the design of a mobile audio device; such as a battery powered headphone amplifier. Although traditionally an analogue product, headphone amplifiers are now systems that contain both analogue and digital components. The analogue components are an analogue amplifier, an amplifying Digital to Analogue Converter (DAC) that generates the analogue signal for the headphone, a Phase-Locked Loop (PLL) that generates a precise clock for the DAC, and a USB-PHY to receive digital data. The digital component is typically a processor that executes the following tasks:

- A USB component that communicates with a mobile phone or tablet to receive audio data. Software on the phone will decide as to what audio is played.
- A DSP component that will perform some operations on the data; this may, for example, be sample rate conversion, converting sample formats, or performing sound enhancement.
- An audio delivery component, that typically drives digital data to the DAC and a base-line clock to the PLL.

Although a sizeable fraction of the power gets used by the USB PHY and the analogue amplifier, designing the software to be low power has a measurable effect on the life time of the battery. This optimization is at present very much a manual effort.

The three software components have different characteristics in terms of their energy requirements:

- The USB component is heavily optimised, and only little room is available for energy optimization; mostly in the form of deploying an extra core and thereby allowing the USB core to be slowed down and scale its voltage.

- The Digital Signal Processing component can be subjected to a variety of optimizations, in particular parallelization and precision trade-offs.
- The sound delivery component is likely to benefit from I/O parallelism and scheduling; although the current component has been hand crafted, clearer components could be written that could be subject to scheduling.

Manual optimization of the system as a whole shows an interesting set of challenges:

- Voltage and frequency scaling applies to the system as a whole; as such, the system that requires the highest frequency sets a limit for the system as a whole. In this particular case study, the system is likely to be limited by the frequency required by the USB software.
- The Digital Signal Processing component and the I/O component are likely to be opposite ends in how power saving is to be applied. From experience, it is best for the DSP to be parallelised (and reduce its frequency at the cost of a few extra threads), whereas the I/O could be combined into a single thread, because each of the tasks is very little work. Not combining them would be expensive and not save power.
- The system as a whole comprises components that work on different sorts of data. The analogue delivery component works with 24 bit data values stored in the upper 24 bits; the DSP component way work on similar values or may work on high frequency 1-bit DAC values that use all 32 bits in a word, and the USB component may be processing packed data (all 32 bits are used), unpacked data (with the data in the top 24 bits), or 1-bit DAC data (all 32-bits used). There may be value in scheduling those tasks or locating those tasks on appropriate threads and cores.

Hence, this case is interesting because it will enable optimization strategies to be explored individually, but there is also insight to be gained from a whole systems perspective.

This system should be optimised primarily for dynamic current consumption. Static current is not relevant since the owner of the system will switch it off when they are not listening to music.

4.4 Robot and motor control

In this area, we focus on driving small motors, where computation is a significant component of system power consumption; in particular we consider:

- a multi-axis system that requires the logic to control the positioning of a multitude of (small) motors;

- motors where the peak power consumption is in the order of no more than a few Watts each;
- motors where the standby consumption is likely to be less than a Watt each.

This makes it worthwhile to reduce the power consumption of the digital part; in particular if the system as a whole spends sufficient time idling. The system typically comprises a motor, a transistor bridge to drive each end of the motor, an ADC to measure current on each end, and a processor that executes the following tasks:

- A PWM driver that generates a Pulse Width Modulated signal for each end of the motor. A PWM signal is a digital approximation of the analogue signal, and it is used to drive the transistors in the bridge. The PWM signal may either be generated by a special hardware component (as found on system-on-chip motor control devices), or by a software component (such as found on software controlled devices). The latter consumes more resources, but gives finer-grained control to the programmer.
- A Digital Signal Processing component that given
  - the currents as measured by the Analogue to Digital converters, and
  - the desired torque

  control the PWM values as applied to the motor. This can be as simple as a series of matrix transformations.
- A controller component that controls the speed and position of each of the motors; this can be as simple as a Proportional Integral Differential controller (PID) which comprises only a multiplications and additions.
- A multi-axis controller that computes the desired positions and speeds of each motor, in order to create a single smooth motion.

These components may be distributed over the system in order to build a modular and extensible system. In that case there will be one digital processor for each motor in a multi-axis system that executes the first three of the software components. A single central processor will implement the final component.

In terms of optimizations, different techniques will apply to different parts of the system. If the system is designed in a distributed manner, then voltage and frequency scaling can be applied individually to the various distributed components of the system. At present, no effort has been spent on optimising the software stack, as it has only been applied to high powered motors.

Optimizations are expected to be valuable across the software components, as there is an imbalance in computation and IO requirements:

- If the PWM driver is implemented in software, then it has very little work except for making sure that signals are switched at precise times. As such, there is scope to schedule this by parallelising the IO with a different task.

- The Digital Signal Processing component performs a reasonable amount of computation, and does so in lock step with the PWM driver and ADC samples. As such, it may be able to shift work from this task into the PWM task (or combine them and then parallelise them) in order to create a balanced workload that can run at a reduced frequency.

- The controller represents a very small workload. This poses an interesting problem that occasionally extra work needs to be performed, and the system needs to be sized to allow for that extra work.

- The multi-axis controller is a significant workload that can be optimised by splitting it out and potentially over multiple cores. It has a soft real-time requirement as it executes only to plan activity. This plan is then executed by the other software components. The multi-axis controller traditionally relies on floating point computations that may be optimised into fixed point computations for processors that do not have hardware floating point units.

In addition to the above optimizations, there is an interesting precision trade-off; driving motors with higher precision will reduce their power consumption. So a trade-off is made between the amount of computation performed digitally, and the power consumed in the analogue part. This is likely to be outside the scope of the project.

This system is likely to be running permanently; as such the important metric for optimization is the dynamic current. Peak power is interesting, but is likely to be close to the average power consumption.

4.5 Real-time networking

The purpose of real-time networking is to deliver data at a guaranteed time. The purpose of real time networking ranges from control in industrial production lines, through to the reversing camera of a car and or the sound distribution in a music arena. In some cases very low power nodes are required, for example when designing a sensor network for example.

The real-time network that the project can look at is ones that are based on the IEEE 1722 standard for “Layer 2 Transport Protocol for Time Sensitive Applications in a Bridged Local

Area Network". This protocol was developed as a protocol for Audio and Video delivery, but is also useful for other data, for example sensor and control data in an industrial context.

A real-time networking system typically comprises a networking PHY (we will just assume some sort of Ethernet PHY; probably one of the low-power 100 Mbit PHYs), a digital processor that implements the standard, and one or more IO devices that are controlled or observed through the real-time network. The latter are application dependent, and may be amplifier units for the music arena, or a motor control system in an industrial setting.

The digital processor executes the following tasks:

- Data transmission and reception tasks; these may be bespoke hardware in SOCs, or maybe software on general purpose processors.
- A receive filter
- A MAC layer for reception
- A traffic shaper/MAC layer for transmission
- An implementation of the Stream Reservation Protocol and Media Reservation Protocol (SRP and MRP).
- An implementation of the control protocol defined in IEEE 1722.1
- An implementation of a peer-to-peer clock protocol.
- An implementation of the AVB protocol in the form of a listener and/or a talker.
- A program that communicates with the target logic.

As far as the software is concerned, many components can run as slow as is sensible; the only real time requirements are that the tasks that operate on data (that is the MAC, transmission, reception, and IO layer) have sufficient bandwidth to deal with the average data rate, and the transmission and reception layers have sufficient bandwidth to deal with the peak data rate.

This means that there is scope to run many tasks at a very low frequency, and only a few tasks at high speed. In particular:

- Data transmission and reception tasks; if these are implemented in software, then they must be designed to be able to keep up with the line rate of the physical network medium (100 Mbit, or 10 Mbit for slow systems).

- The receive filter just has to keep up with the line-packet-rate; that is, it must be analysed for being able to parse headers and test headers at line rate (for minimum sized packets that is approximately 150 kPackets per second for a 100 Mbit line rate).
- The MAC layers have to be able to keep up with the average data rate expected on the node; typically well below line rate.
- The SRP, MRP, and 1722.1 control protocols run with time-outs of tens or hundreds of milliseconds, and must be analysed to complete in time for that. They can also run to completion if that is more efficient.
- The peer-to-peer clock protocol expects precise timestamps from the data transmission and reception tasks, but can itself run at a low frequency, needing to complete only in 10s of milliseconds.
- The implementation of the talker and listener have to be able to deal with the average data rate of the payload, excluding headers.

A completed analysis of all tasks will yield minimum speeds that will all be quite low, except for the direct transmit and receive tasks. These tasks may be combined and executed in a single task, or offloaded to a secondary very slow core.

There is little scope to manage precision or hamming distances.

None of these optimizations or analysis have been performed manually; at present the software stack runs at full speed.

This system should be optimised primarily for dynamic current consumption and static current consumption; the peak power demand should be estimated but need not be optimised. The system is likely to have phases of activity, which means that energy use will be a combined version of static and dynamic power figures. Peak performance is important, as we expect fluctuation of power demands when packets arrive or are transmitted (which may happen simultaneously).

5 Assessment and Relation to Future Tasks

In this report we have surveyed many energy-related optimizations that can be obtained by static software improvements. That is we have surveyed ways in which code can be transformed during development or at compile-time to reduce various aspects of energy usage. Different energy metrics were discussed in Deliverable D6.1 of the ENTRA project [Mul13]. As expected, most optimizations are dependent on features of the underlying architecture, instruction-set and the availability of hardware power-saving features. In Section 3 the survey considered optimizations

for a wide variety of platforms, while in the Section 4 we focus on general features of, and optimizations for, the ENTRA project target platform, namely the xCORE family of processors produced by XMOS Ltd.

5.1 Focus areas for future research on optimization (WP4)

While there is a significant overlap between performance optimizations and energy optimizations, there are opportunities for energy optimization where execution time may be unaffected or even degraded. Static execution time optimization has been studied intensively for a long time, whereas energy optimizations that are independent of performance are generally speaking less understood and researched. Such optimizations are especially significant for the ENTRA project.

The opportunities for optimization in the case studies suggest the following focus areas for research in the coming months in WP3 (Analysis) and WP4 (Optimization) of the ENTRA project. Most of the gains are going to be made by choosing an appropriate level of parallelism in the system. In the following paragraphs we summarise the directions and ongoing research tasks.

Scheduling of communication versus computation. A rule of thumb for energy optimization, which is being applied to the case studies, is to ask "how slow can a computation run in order to meet the application's internal and external communication requirements?". To answer this question requires subsidiary analyses to those considering energy directly. The analysis should expose essential dependencies of the communicative structure of an application, and the (probabilistic) rates of communication and computation. The aim is to allow communication and computation to be scheduled optimally, without unnecessary delays and ensuring that time-critical communications are made "just in time". Related optimization principles relate to task location; tasks that communicate frequently should be located as physically close to each other as possible, and threads with similar timing requirements should be on the same core.

Trading off parallelism against other costs. Introducing as much parallelism as possible, and running processors slower as a result, is a principle that can lead to significant energy savings. However, parallelism usually comes with overheads of communication, replication of data, and synchronization delays, as well as expense in manufacture and materials. Thus, analyses of the case studies that allow these trade-offs to be evaluated are important for optimizations based on parallelism. Other aspects of parallelism to be investigated, looking beyond optimization for the xCORE, includes fitting code to what the hardware offers, such as the number of threads in software and hardware. In this work we aim to look also at GPUs as a target as well as the Intel Xeon Phi.

Load balancing. Achieving the above goals for a given application requires detailed understanding of the overall thread behaviour of an application: the distribution over time of the running threads, how active or inactive each thread is, how much work each thread does for each communication with other threads and whether there are workload spikes causing high peak power consumption. Once this is understood the thread structure can be optimized to eliminate bottlenecks and synchronization delays.

Precision management. This is going to be of less immediate value in the chosen case studies, but it is still a valuable tool when managing energy consumption. In certain applications, energy is wasted by computing results that are more precise than required, for example using higher-precision numbers than are needed. Analysing and identifying such cases can lead to energy optimizations by reducing precision. In some cases this is a trade-off against quality; determining what quality is "good enough" is the key to making optimizations.

Various low-level energy improvements. We will also explore low-level optimizations applicable to xCORE and other targets, such as alignment of loops to line boundaries in memory.

5.2 Relation to Dynamic Optimization

As emphasised earlier, this deliverable covers static optimizations, applicable during application development and compilation. The deliverable provides input to Tasks T4.2 and T4.3 of the ENTRA project, which consider a wider range of optimizations, including dynamic optimizations. These could be for example dynamically trading off energy against quality of service (T4.2) by modifying precision or processor speed, or energy-aware scheduling (T4.3). We note here that static optimization and code design can enable more effective dynamic optimizations, for example building into the code parameters that control quality; such parameters can be dynamically adjusted to suit the perceived quality needs. We also note that energy modelling (Work Package WP2) is critical to both static and dynamic optimization.

5.3 Relation to Energy-Aware Software Engineering

Although this deliverable does not cover tool support, it provides input to Work Package WP1 (Energy-Aware Software Engineering) and in particular Task T1.1 (Energy-Aware Tools). Initially, tools will focus on energy transparency rather than automatic optimization. Energy transparency tools will help the developer to obtain information about the energy consumption of the program, thus assisting the developer in identifying opportunities for manual optimization.

They will also support the evaluation of optimizations. Other tools, needed to support the optimizations discussed in Section 4, are those providing information about dependencies within the program especially those relating to communication, timing, and the distribution of parallel tasks.

References

[AAB09] José L. Ayala, David Atienza, and Philip Brisk. Thermal-aware data flow analysis. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 613–614, New York, NY, USA, 2009. ACM.

[Aga09] Anant; R. Agarwal. Using Code Perforation to Improve Performance, Reduce Energy Consumption, and Respond to Failures. Technical report, MIT, September 2009.

[ALV05] José Luis Ayala and Marisa López-Vallejo. Compiler-driven power optimizations in the register file of processor-based systems. In Luca Benini, Ulrich Kremer, Christian W. Probst, and Peter Schelkens, editors, *Power-aware Computing Systems*, volume 05141 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2005.

[AVLV03] José L. Ayala, Alexander Veidenbaum, and Marisa López-Vallejo. Power-Aware Compilation for Register File Energy Reduction. *International Journal of Parallel Programming*, 31(6):451–467, December 2003.

[Aze06] Naeem Zafar Azeemi. Exploiting parallelism for energy efficient source code high performance computing. In *Industrial Technology, 2006. ICIT 2006. IEEE International Conference on*, pages 2741–2746, 2006.

[BA06] Sorav Bansal and Alex Aiken. Automatic generation of peephole superoptimizers. *ACM SIGOPS Operating Systems Review*, 40(5):394, October 2006.

[BC10] Woongki Baek and Trishul M. Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. *SIGPLAN Not.*, 45(6):198–209, June 2010.

[BCG+13] Gilles Barthe, Juan Manuel Crespo, Sumit Gulwani, César Kunz, and Mark Marron. From Relational Verification to SIMD Loop Synthesis. In Alex Nicolau, Xiaowei Shen, Saman P. Amarasinghe, and Richard Vuduc, editors, *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13*, pages 123–134. ACM, 2013.

[BCVF06] Martin Brain, Tom Crick, Marina De Vos, and John Fitch. Toast: Applying answer set programming to superoptimisation. In *Int. Conf. Logic Programming*, pages 270–284, 2006.

[BWM+09] A. Banerjee, P.T. Wolkotte, R.D. Mullins, S.W. Moore, and G. J M Smit. An energy and performance exploration of network-on-chip architectures. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 17(3):319–329, 2009.

[CDYW05] Juan Chen, Yong Dong, Xue-jun Yang, and Dan Wu. A compiler-directed energy saving strategy for parallelizing applications in on-chip multiprocessors. In *Parallel and Distributed Computing, 2005. ISPDC 2005. The 4th International Symposium on*, pages 147–154, 2005.

[CGL10] Swarat Chaudhuri, Sumit Gulwani, and Roberto Lublinerman. Continuity analysis of programs. *SIGPLAN Not.*, 45(1):57–70, January 2010.

[CM10] Sangyeun Cho and Rami G. Melhem. On the interplay of parallelization, program performance, and energy consumption. *IEEE Trans. Parallel Distrib. Syst.*, 21(3):342–353, March 2010.

[DMCN12] J. Diaz, C. Munoz-Caro, and A. Nino. A survey of parallel programming models and tools in the multi and many-core era. *Parallel and Distributed Systems, IEEE Transactions on*, 23(8):1369–1386, 2012.

[Edw11] C. Edwards. Lack of Software Support marks the Low Power Scorecard at DAC. ElectronicsWeekly (http://www.electronicsweekly.com), No. 2472, 15-21 June 2011.

[EG13] K. Eder and N. Grech, editors. *Common Assertion Language*. ENTRA Project: Whole-Systems Energy Transparency (FET project 318337), November 2013. Deliverable 2.1, http://entraproject.eu.

[GJTV12] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. *ACM SIGPLAN Notices*, 47(6):62, 8 2012.

[GK92] Torbjörn Granlund and Richard Kenner. Eliminating branches using a superoptimizer and the GNU C compiler. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation - PLDI '92*, pages 341–352, New York, New York, USA, 1992. ACM Press.

[GKD11] Mark Gebhart, Stephen W. Keckler, and William J. Dally. A compile-time managed multi-level register file hierarchy. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, pages 465–476, New York, NY, USA, 2011. ACM.

[GSB95] David A. Garza-Salazar and Wim Böhm. Reducing communication by honoring multiple alignments. In *Proceedings of the 9th international conference on Supercomputing*, ICS '95, pages 87–96, New York, NY, USA, 1995. ACM.

[HK03] Chung-Hsing Hsu and Ulrich Kremer. The design, implementation, and evaluation of a compiler algorithm for cpu energy reduction. *SIGPLAN Not.*, 38(5):38–48, May 2003.

[HPH+02] T. Heath, E. Pinheiro, J. Hom, U. Kremer, and R. Bianchini. Application transformations for energy and performance-aware device management. In *Parallel Architectures and Compilation Techniques, 2002. Proceedings. 2002 International Conference on*, pages 121–130, 2002.

[HRB88] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 35–46, New York, NY, USA, 1988. ACM.

[HSC+11] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Dynamic knobs for responsive power-aware computing. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS XVI, pages 199–212, New York, NY, USA, 2011. ACM.

[JOA+09] Timothy M. Jones, Michael F. P. O'Boyle, Jaume Abella, Antonio González, and Oğuz Ergin. Energy-efficient register caching with compiler assistance. *ACM Trans. Archit. Code Optim.*, 6(4):13:1–13:23, October 2009.

[KA09a] Vijay Anand Korthikanti and Gul Agha. Analysis of parallel algorithms for energy conservation in scalable multicore architectures. In *Proceedings of the 2009 International Conference on Parallel Processing*, ICPP '09, pages 212–219, Washington, DC, USA, 2009. IEEE Computer Society.

[KA09b] Vijay Anand Korthikanti and Gul Agha. Energy-bounded scalability analysis of parallel algorithms. In *Technical Report, Department of Computer Science, University of Illinois at Urbana Champaign*, 2009.

[KA10] Vijay Anand Korthikanti and Gul Agha. Towards optimizing energy costs of algorithms for shared memory architectures. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, SPAA '10, pages 157–165, New York, NY, USA, 2010. ACM.

[KA11] Vijay Anand Korthikanti and Gul Agha. Energy-performance trade-off analysis of parallel algorithms for shared memory architectures. *Sustainable Computing: Informatics and Systems*, 1(3):167 – 176, 2011. ¡ce:title¿Theoretical aspects of Sustainable Computing¡/ce:title¿.

[KE13] S. Kerrison and K. Eder. Energy modelling and optimisation of software for a hardware multi-threaded embedded microprocessor. Technical report, University of Bristol, June 2013.

[KVI02] Mahmut Kandemir, N. Vijaykrishnan, and Mary Jane Irwin. Compiler optimizations for low power systems. In Robert Graybill and Rami Melhem, editors, *Power aware computing*, pages 191–210. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[LG13] P. López-García, editor. *A General Framework for Resource Consumption Analysis and Verification*. ENTRA Project: Whole-Systems Energy Transparency (FET project 318337), November 2013. Deliverable 3.1, http://entraproject.eu.

[LMD+04] Markus Lorenz, Peter Marwedel, Thorsten Dräger, Gerhard Fettweis, and Rainer Leupers. Compiler based exploration of dsp energy savings by simd operations. In *Proceedings of the 2004 Asia and South Pacific Design Automation Conference*, ASP-DAC '04, pages 838–841, Piscataway, NJ, USA, 2004. IEEE Press.

[LY07] Xuanhua Li and D. Yeung. Application-level correctness and its impact on fault tolerance. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 181–192, 2007.

[Mas87] Henry Massalin. Superoptimizer - A Look at the Smallest Program. *ACM SIGARCH Computer Architecture News*, pages 122–126, 1987.

[MG08] Simon Moore and Daniel Greenfield. The next resource war: computation vs. communication. In *Proceedings of the 2008 international workshop on System level interconnect prediction*, SLIP '08, pages 81–86, New York, NY, USA, 2008. ACM.

[MLN+06] Madhu Mutyam, Feihui Li, Vijaykrishnan Narayanan, Mahmut Kandemir, and Mary Jane Irwin. Compiler-directed thermal management for vliw functional units. *SIGPLAN Not.*, 41(7):163–172, June 2006.

[MRR11a] Sasa Misailovic, Daniel M. Roy, and Martin C. Rinard. Probabilistic and statistical analysis of perforated patterns. Technical Report MIT-CSAIL-TR-2011-003, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, January 2011.

[MRR11b] Sasa Misailovic, Daniel M. Roy, and Martin C. Rinard. Probabilistically accurate program transformations. In *Proceedings of the 18th international conference on Static analysis*, SAS'11, pages 316–333, Berlin, Heidelberg, 2011. Springer-Verlag.

[MSHR10] Sasa Misailovic, Stelios Sidiroglou, Henry Hoffmann, and Martin Rinard. Quality of service profiling. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 25–34, New York, NY, USA, 2010. ACM.

[Mul13] H. Muller, editor. *Metrics and Case Studies*. ENTRA Project: Whole-Systems Energy Transparency (FET project 318337), November 2013. Deliverable 6.1, http://entraproject.eu.

[NR07] Huu Hai Nguyen and Martin Rinard. Detecting and eliminating memory leaks using cyclic memory allocation. In *Proceedings of the 6th international symposium on Memory management*, ISMM '07, pages 15–30, New York, NY, USA, 2007. ACM.

[OKC11] O. Ozturk, M. Kandemir, and G. Chen. Compiler-directed energy reduction using dynamic voltage scaling and voltage islands for embedded systems. *Computers, IEEE Transactions on*, 62(2):268–278, 2011.

[PKVI00] A. Parikh, M. Kandemir, N. Vijaykrishnan, and M.J. Irwin. Instruction scheduling based on energy and performance constraints. In *VLSI, 2000. Proceedings. IEEE Computer Society Workshop on*, pages 37–42, 2000.

[RHMS10] Martin Rinard, Henry Hoffmann, Sasa Misailovic, and Stelios Sidiroglou. Patterns and statistical analysis for understanding reduced resource computing. *SIGPLAN Not.*, 45(10):806–821, October 2010.

[Rin06] Martin Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *Proceedings of the 20th annual international conference on Supercomputing*, ICS '06, pages 324–334, New York, NY, USA, 2006. ACM.

[RJ97] Kaushik Roy and Mark C. Johnson. Software Design for Low Power. In Wolfgang Nebel and Jean P. Mermet, editors, *Low Power Design in Deep Submicron Electronics*, volume 337, pages 433–460. Kluwer Academic, 1997.

[RK08] C.C. Rolf and K. Kuchcinski. State-copying and recomputation in parallel constraint programming with global constraints. In *Parallel, Distributed and Network-Based Processing, 2008. PDP 2008. 16th Euromicro Conference on*, pages 311–317, 2008.

[SAA10] Mohamed M. Sabry, José L. Ayala, and David Atienza. Thermal-aware compilation for system-on-chip processing architectures. In *Proceedings of the 20th symposium on Great lakes symposium on VLSI*, GLSVLSI '10, pages 221–226, New York, NY, USA, 2010. ACM.

[SB13] Yakun Sophia Shao and David Brooks. Energy characterization and instruction-level energy model of Intel's Xeon Phi processor. In *International Symposium on Low Power Electronics and Design (ISLPED)*, pages 389–394. IEEE, November 2013.

[SCO+07] Seung Woo Son, Guangyu Chen, O. Ozturk, M. Kandemir, and A. Choudhary. Compiler-directed energy optimization for parallel disk based systems. *Parallel and Distributed Systems, IEEE Transactions on*, 18(9):1241–1257, 2007.

[SDF+11] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. Enerj: Approximate data types for safe and general low-power computation. *SIGPLAN Not.*, 46(6):164–174, June 2011.

[SDMHR11] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 124–134, New York, NY, USA, 2011. ACM.

[SHMC12] Dongrui She, Yifan He, Bart Mesman, and Henk Corporaal. Scheduling for register file energy minimization in explicit datapath architectures. In Wolfgang Rosenstiel and Lothar Thiele, editors, *DATE*, pages 388–393. IEEE, 2012.

[SKL01] Dongkun Shin, Jihong Kim, and Seongsoo Lee. Intra-task voltage scheduling for low-energy hard real-time applications. *Design Test of Computers, IEEE*, 18(2):20–30, 2001.

[SSA13] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *Architectural Support for Programming Languages and Operating Systems*, page 305, New York, New York, USA, 2013. ACM Press.

[SSCK14] Wook Song, Nosub Sung, Byung-Gon Chun, and Jihong Kim. Reducing energy consumption of smartphones using user-perceived response time analysis. In *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications*, HotMobile '14, pages 20:1–20:6, New York, NY, USA, 2014. ACM.

[SyTD94] C.-L. Su, Chi ying Tsui, and A.M. Despain. Low power architecture design and compilation techniques for high-performance processors. In *Compcon Spring '94, Digest of Papers.*, pages 489–498, 1994.

[Wei81] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[ZMKR12] Zeyuan Allen Zhu, Sasa Misailovic, Jonathan A. Kelner, and Martin Rinard. Randomized accuracy-aware program transformations for efficient approximate computations. *SIGPLAN Not.*, 47(1):441–454, January 2012.

Attachments

Attachment D4.1.1

Study of Possible Static Power Reduction due to Temperature Hot Spot Reduction provided by Uniform Register Utilization

Zorana Banković. Technical Report. IMDEA Software Institute, Madrid, Spain

Study of Possible Static Power Reduction due to Temperature Hot Spot Reduction provided by Uniform Register Utilization

Zorana Banković

IMDEA Software Institute, Madrid, Spain

1 The Estimation of Static Power Savings

In order to evaluate possible savings, first we have to establish a thermal model of the underlying system. In general, these models are based on thermal resistance networks, which are analogue to DC circuits where the voltage of each node is equivalent to the temperature, and the current is equivalent to the heat flow between two nodes, i.e. $q = K \cdot \Delta T$, where K is thermal conductivity (thermal resitance is $1/K$), which depends on the technology and the geometry of the component. Thus, these networks can be solved in the same way as DC circuits.

In this calculation we will take a very coarse-grain model, where it is assumed that a whole register can be modelled as a unique thermal resistance, which is the same for each register. Furthermore, we will take a set of 12 registers, since each logical core in XMOS has its one register set of 12 registers. Initially we will assume that different register sets belonging to different register cores are situated far enough, so they do not affect each other. However, thermal diffusion cannot be neglected, so we have to add a resistor between each two points to account for this phenomenon. Having in mind that thermal conductivity decreases linerly with distance, and assuming that the distance between each two nodes is eqaul, the conductivity between the first and the third register will be $K/2$, between the first and the forth will be $K/3$, etc. This is depicted in figure 1, where we can observe only the resistors which stand for the diffusion between the first registers and the rest. However, it is important to point out that thermal diffusion exists between each two registers. Thus, the complete model will contain all these elements, but we do not include it here due to its complexity.

The model contains values $q_1, ..., q_{12}$, which stand for the heat flow produced by register accesses. We assume that the heat flow is directly proportional to the number of register accesses, i.e. $q_i = a_i \cdot C$, where a_i is the number of accesses to the register, and C is the constant which is the same for all. Thus, having the complete model, we can solve it using one of the ways for solving DC circuits. Here we have used matrix inverse method [2]. However, since we do not have enough technology parameters, we cannot know the absolute values of the thermal conductivities, nor we know the absolute values of the heat flows. Thus, the solution of the presented model can give us only the relation between the temperature increase of each register. This is presented in the following table 1. The first column states which registers are accessed uniformly, while the rest contain

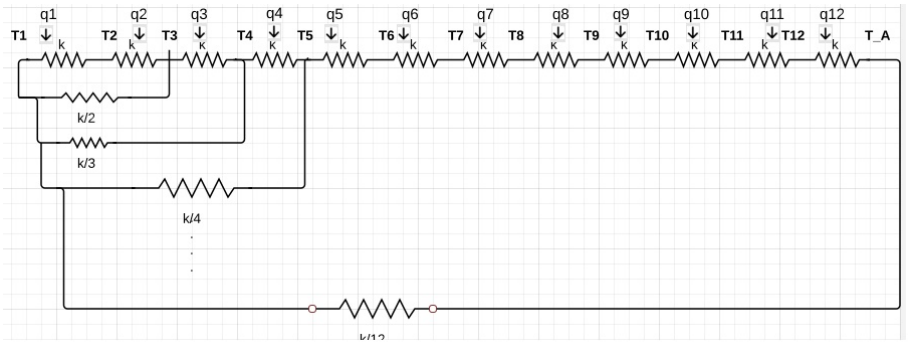

Fig. 1. Reduced Thermal Model of 12 Registers

coefficients C_{ij} used to calculate the temperature of each register according to the formula: $T_{ij} = T_A + C_{ij} \cdot \Delta T$, where T_A is the ambient temperature, i.e. the temperature of the rest of the chip, while ΔT is the temperature increase due to register access.

Table 1.

Accessed reg.	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}	T_{11}	T_{12}
1	2.45	1.4	1.38	1.3	1.27	1.24	1.2	1.16	1.13	1.09	1.02	1.0
1,2	1.93	1.85	1.37	1.33	1.28	1.24	1.2	1.17	1.13	1.09	1.02	1.0
1,2,3	1.74	1.68	1.63	1.33	1.28	1.24	1.2	1.17	1.13	1.09	1.02	1.0
1,2,3,4	1.64	1.6	1.56	1.52	1.28	1.24	1.2	1.17	1.13	1.09	1.02	1.0
1,2,3,4,5	1.56	1.54	1.51	1.47	1.44	1.24	1.2	1.17	1.13	1.09	1.02	1.0
1,2,3,4,5,6	1.51	1.48	1.46	1.44	1.41	1.37	1.2	1.17	1.13	1.09	1.02	1.0
1,2,3,4,5,6,7	1.47	1.45	1.43	1.41	1.38	1.35	1.32	1.18	1.14	1.09	1.02	1.0
1,2,3,4,5,6,7,8	1.42	1.41	1.4	1.38	1.36	1.33	1.3	1.27	1.14	1.09	1.02	1.0
1,2,3,4,5,6,7,8,9	1.4	1.38	1.37	1.35	1.33	1.31	1.28	1.25	1.22	1.09	1.03	1.0
1,2,3,4,5,6,7,8,9,10	1.4	1.34	1.33	1.32	1.3	1.29	1.26	1.24	1.21	1.17	1.03	1.0
1,2,3,4,5,6,7,8,9,10,11	1.33	1.32	1.3	1.3	1.28	1.27	1.25	1.22	1.2	1.16	1.09	1.0
1,2,3,4,5,6,7,8,9,10,11,12	1.31	1.3	1.29	1.28	1.26	1.25	1.23	1.21	1.19	1.15	1.09	1.08

We will now perform a simulation with different values of T_A and ΔT. ΔT takes values from 0 to 20°C, which means high register usage. On the other hand, to be able to estimate power savings, we need to estimate the leakage current. This information is provided by XMOS [1], where leakage dependence on the temperature is depicted. The graph which depicts typical value is taken and fitted with the following formula:

$$I_{leak} = 0.085 \cdot T^2 - 1.28 \cdot T + 272.42 \tag{1}$$

The average chip temperature is calculated as average of T_A and $T_1, ..., T_{12}$, and the leakage current is calculated using the previous formula. Since the voltage does not change throughout the whole process, and given that $P = U \cdot I$, the current is the one that determines the amount of the saved energy.

In order to illustrate the findings of this experiment, we will illustrate the difference of two extreme cases: in the first case only the first register in the set is used, while in the second case, all registers are uniformly accessed. The power savings for different versions of T_A and ΔT are depicted in figure 2. As we can observe, maximal achievable savings are around 3%, which is not very significant, having in mind that static power makes 30% of total power, which gives total savings around 0.9%. However, if we take a look at table 1, we can observe that in the previous extreme cases the difference between the hotspots can become significant, i.e. $(2.45 - 1.31) \cdot \Delta T = 1.14\Delta T$, which can affect significantly on cooling power. For this reason, according to the findings of this experiment, the methods based on making register access more uniform from the point of view of power and energy savings become more significant when the cooling power becomes significant part of total power, such as in general purpose systems and especially in data centers.

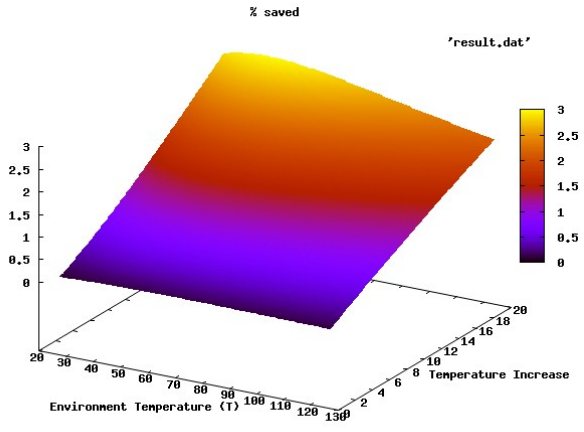

Fig. 2. Power Savings in Different Scenarios

References

1. XMos Ltd. Estimating power consumption for xs1-g devices, may 2012.
2. Robert E. Simons. Using a matrix inverse method to solve a thermal resistance network. http://www.electronics-cooling.com/2009/05/

using-a-matrix-inverse-method-to-solve-a-thermal-resistance-network/, may 2009.

Attachment D4.1.2

From Relational Verification to SIMD Loop Synthesis.

Published in the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13.

From Relational Verification to SIMD Loop Synthesis

Gilles Barthe[1] Juan Manuel Crespo[1] Sumit Gulwani[2] César Kunz[1,3] Mark Marron[1]

[1]IMDEA Software Institute, [2]Microsoft Research, [3]Technical University of Madrid

{gilles.barthe, juanmanuel.crespo, cesar.kunz, mark.marron}@imdea.org, sumitg@microsoft.com

Abstract

Existing pattern-based compiler technology is unable to effectively exploit the full potential of SIMD architectures. We present a new program synthesis based technique for auto-vectorizing performance critical innermost loops. Our synthesis technique is applicable to a wide range of loops, consistently produces performant SIMD code, and generates correctness proofs for the output code. The synthesis technique, which leverages existing work on relational verification methods, is a novel combination of deductive loop restructuring, synthesis condition generation and a new inductive synthesis algorithm for producing loop-free code fragments. The inductive synthesis algorithm wraps an optimized depth-first exploration of code sequences inside a CEGIS loop. Our technique is able to quickly produce SIMD implementations (up to 9 instructions in 0.12 seconds) for a wide range of fundamental looping structures. The resulting SIMD implementations outperform the original loops by 2.0×-3.7×.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; D.3.4 [*Programming Languages*]: Processors-Optimization; C.1.1 [*Single Data Stream Architectures*]: VLIW architectures

Keywords Program Vectorization, Program Synthesis, Deductive Synthesis, Inductive Synthesis, Relational Program Verification

1. Introduction

Single Instruction Multiple Data (SIMD) instructions sets (such as SSE on x86 or NEON on ARM) provide high throughput and power efficient data-parallel operations. These operations can process 128 bits in a single instruction and can often do so in the same number of cycles (and power usage) needed to process a single 32 bit value via the standard ALU execution path. These features have proven invaluable in accelerating multimedia and high performance computing applications, and are critical to achieving both good application performance and battery life in many mobile computing environments. Despite these advantages and their proven value in practice, the use of SIMD operations has been limited to a relatively small set of (often hand optimized) applications. Extending these benefits to a wider range of programs via automatic compiler vectorization has, in practice, been limited by three major challenges: the presence of pointers, sub-optimal data layout, and complex data driven control flow. In this paper we explore a new approach to auto-vectorization that is intended to address the last two of these challenges. This approach allows us to produce efficient SIMD implementations for many loops that are present in foundational libraries such as the STL for C++ or the BCL for C#.

PPoPP'13, February 23–27, 2013, Shenzhen, China.


```
//Simple widget struct with a tag and a score value
struct { int tag; int score; } widget;

int exists(widget* vals, int len, int tv, int sv) {
  for (int i = 0; i < len; ++i) {
    int tagok = vals[i].tag == tv;
    int scoreok = vals[i].score > sv;
    int andok = tagok & scoreok;
    if (andok) return 1;
  }
  return 0;
}
```

Figure 1. Initial Loop.

```
int exists_sse(widget* vals, int len, int tv, int sv) {
  m128i vectv = [tv, tv, tv, tv];
  m128i vecsv = [sv, sv, sv, sv];

  int i = 0;
  for (; i < (len - 3); i += 4) {
    m128i blck1i = load_128(vals + i);
    m128i blck2i = load_128(vals + i + 2);

    int tvswizzle = SHF_ORDER(0, 2, 0, 2);
    int svswizzle = SHF_ORDER(1, 3, 1, 3);

    m128i tagvs = shuffle_i32(blck1i, blck2i, tvswizzle);
    m128i scorevs = shuffle_i32(blck1i, blck2i, svswizzle);

    m128i cmprl = cmpeq_i32(vectv, tagvs);
    m128i cmprh = cmpgt_i32(vecsv, scorevs);
    m128i cmpr = and_i128(cmprl, cmprh);

    int match = !allzeros(cmpr);
    if (match) return 1;
  }

  for (; i < len; i++) {
    int tagok = vals[i].tag == tv;
    int scoreok = vals[i].score > sv;
    if (tagok & scoreok) return 1;
  }
  return 0;
}
```

Figure 2. SIMD Implementation.

Motivating Example. Consider the program fragment in Figure 1, which consists of a loop that traverses an array of `widget` structs (of length `len`). The loop body checks if the values in the `tag` and `score` fields satisfy certain properties and if so returns 1 immediately. If no such `widget` is found then 0 is returned.

$$
\begin{array}{rrcl}
(variables) & & & a : \text{Array} \mid i, x : \text{Int32} \mid s : \text{Struct} \mid \mathbf{v} : \mathbf{Vector} \\
(fields) & & & f \in \text{Field} \\
(constants) & c & ::= & \mathbb{Z} \mid \mathbf{SHF\_ORDER}(\mathbf{c},\mathbf{c},\mathbf{c},\mathbf{c}) \\
(expr) & e & ::= & c \mid x \mid a[i] \mid e.f \mid x \circ x, \text{ where } \circ \in \{+,=,\&,\ldots\} \mid \mathbf{allzeros}(\mathbf{v}) \\
(vector\ expr) & \mathbf{ve} & ::= & \mathbf{v} \mid \langle \mathbf{e},\mathbf{e},\mathbf{e},\mathbf{e} \rangle \mid \mathbf{load\_128}(\mathbf{a},\mathbf{i}) \mid \mathbf{shuffle\_i32}(\mathbf{v},\mathbf{v},\mathbf{c}) \\
 & & & \mid \mathbf{op}(\mathbf{v},\mathbf{v}), \text{ where } \mathbf{op} \in \{\mathbf{add\_i32}, \mathbf{cmpeq\_i32}, \mathbf{and\_i128}, \ldots\} \\
(stmts) & st & ::= & x := e \mid a[i] := x \mid e.f := x \mid \mathsf{skip} \mid \mathbf{v} := \mathbf{ve} \mid \mathbf{store\_128}(\mathbf{a},\mathbf{i},\mathbf{v}) \\
(block) & b & ::= & st \mid b; b \mid \mathsf{if}\ x\ \mathsf{then}\ b\ \mathsf{else}\ b \\
(flowblock) & f & ::= & b \mid \mathsf{return}\ x \mid \mathsf{break} \mid f; f \mid \mathsf{if}\ x\ \mathsf{then}\ f\ \mathsf{else}\ f \\
(loop) & \ell & ::= & \mathsf{for}\ i := e;\ i \bowtie e';\ i = i \pm c;\ \mathsf{do}\ f, \text{where } \bowtie \in \{=,\neq,<,>,\leq,\geq\} \wedge e' : \text{Int32} \wedge e' \text{ is invariant in } f \\
(fragment) & \delta & ::= & f; \ell; \boldsymbol{\ell*}; f
\end{array}
$$

Figure 3. Program Fragment Language and SIMD extensions (in bold)

This loop contains two major challenges from the viewpoint of automatic vectorization. First is that since the loop can exit on any iteration (*i.e.* `return 1`) the loop carries a control flow dependence on all previous iterations. Second is the fact that the data is poorly laid out for SIMD processing – it is in an *array of structs*. Thus, doing a block load from the array will get a mixture of the `tag` and `score` fields. Since these fields are processed differently in the loop body (`tag == tv` vs. `score > sv`) the mixture prevents the direct use of SIMD operations (which apply the same operation to each value). Thus, this loop body does not fit into a standard vectorization template form. Attempting to write a compiler that recognizes and transforms this loop appropriately based on a set of pattern matching rules is unattractive from both an implementation effort and complexity standpoint.

Despite these complications it is possible to construct an efficient SIMD implementation using the SSE instructions found in x86 processors (see Figure 2). The program first loads two data blocks of 128 bits each (two `widget` structs per load) from the array via the `load_128` operation. The SSE implementation handles the array-of-struct issue by *swizzling* [14] the four `tag` values into one SSE register (`tagvs`) and the four `score` fields into a second SSE register (`scorevs`). This is done by computing two swizzle masks, `tvswizzle` and `svswizzle`, and using them to control how the data that was loaded from the array is unpacked by the `shuffle_i32` operations. The `tvswizzle` mask indicates that the 0^{th} and 2^{nd} entries, which contain the `tag` fields, should be loaded from `blck1i` and `blck2i`, and these four values should be placed into `tagvs`. Similarly the 1^{st} and 3^{rd} values, which contain the `score` fields, should be placed into `scorevs`. Once these values are unpacked it is then simple to apply the appropriate SIMD equality (`cmpeq_i32`) and greater than (`cmpgt_i32`) operations to compare the four `tag` fields and the four `score` fields. These comparison operations produce bitmasks in the result vector, all 1's if the test result is true and all 0's if the result is false, for each 32 bit value. The results of these comparisons are then bitwise anded in one step via the `and_i128` operation. The final test (`!allzeros(cmpr)`) checks if any of the `widgets` processed satisfy the constraints and if one does then the `match` value will be 1. Since the original loop simply returns on finding a matching `widget` the SSE loop returns 1 if any of the four `widgets` being processed match (*i.e.* the `allzeros` value is 0). The resulting SSE implementation outperforms the simple loop by well over a factor of $2\times$ for large numbers of iterations and is 25% faster even on small iteration counts.

There are a number of challenges present when designing a system to automatically vectorize loops, such as the one in Figure 1. The first challenge is structuring the vectorization algorithm such that it is *applicable* to a wide range of loops and variations in how they are implemented [21]. This is critical to ensuring that the auto-vectorization is consistently able to find optimized implementations for loops in the input programs and thus improve performance in practice. The next challenge is that the process of vectorizing code often adds complexity and overhead. In order to avoid slowing down the program instead of speeding it up, it is useful to be able to predict if (and when) the SIMD implementation will *reliably improve* the performance of the program relative to the initial implementation. Finally, a fundamental issue in any compiler optimization is *correctness*. Since compiler bugs may introduce errors into every program that is compiled, it is critical to ensure that the resulting SIMD code is equivalent to the input program.

Contributions. To construct an auto-vectorization algorithm that achieves the desired *applicability*, *reliable improvement*, and *correctness* objectives, this paper makes the following contributions:

- A new methodology for program optimization (Section 4) based on a novel combination of: *deductive* rewriting of loop and control-flow structures, *inductive* synthesis of the desired code blocks, and a novel construction based on relational program verification to connect the deductive and inductive steps.
- The methodology is applied to the problem of auto-vectorization of irregular loops that have sub-optimal data layouts and complex data driven control flow. In particular we look at library code from the C++ STL or the C# Base Class Libraries.
- An efficient technique for inductive synthesis of loop-free code fragments, based on a novel combination of concrete program execution, bounded search techniques, and symbolic counter example generation methods (Section 5).
- An experimental evaluation of the auto-vectorizer on a set of challenge loops and real-world applications (Section 7). The results show that the technique performs well in practice: producing SIMD implementations which outperform the original implementations by $2.0\times$-$3.7\times$. We also apply the technique to vectorize loops in the SPEC 483.Xalan benchmark to obtain a 5.5% reduction in runtime.

2. Relational Verification

We begin by reviewing *relational verification* [8, 42] and explain how this technique can be used to reason about the equivalence of two implementations of a loop. We will then introduce two novel forms of *equivalence relations* on program variables for showing the equivalence of a scalar and a vectorized loop.

2.1 Relational Verification Background

The key insight in relational verification is that given two similar programs one does not need to know the exact functionality of the two programs in order to show that they are equivalent. It is sufficient to show that, at the appropriate *synchronization points* during their execution, the states of the two programs are equivalent under some relation. Consider the loops:

```
int sum = 0;
for(int i = 0; i < n; i++)
  sum += i;
```

```
int j = -1;
int sum = 0;
for(int i = 0; i < n; i++) {
  j++;
  sum += j;
}
```

We begin by renaming any variables v which appear in both programs as $v_{\langle 1 \rangle}$ for the value of the variable in the first program on the left and $v_{\langle 2 \rangle}$ for the value of the variable in second program on the right. After this renaming then the equality relation for the states of the two programs is $i_{\langle 1 \rangle} = i_{\langle 2 \rangle} \wedge j = i_{\langle 1 \rangle} - 1 \wedge sum_{\langle 1 \rangle} = sum_{\langle 2 \rangle}$.

Using this relationship we can show these two loops compute the same value. We begin by checking that, when the loop iterations are run in lockstep, at every iteration the states of the programs are equivalent under the relation. Once we have shown that the equivalence relations hold at every loop iteration we can show that they hold after the exit of the loop as well. Thus, we can generate a proof that the two loops compute the same values for the final sums and are observationally equivalent, i.e. $sum_{\langle 1 \rangle} = sum_{\langle 2 \rangle}$.

A critical step in this process is obtaining suitable equality relations. Techniques for obtaining some of these relations, particularly relating to loop structure and conditional control flow, have been developed in previous work [4]. Loop splitting and unrolling are standard transformations which make latent data-parallelism in the loop body more easily exploitable. As SIMD operations operate on k values at a time we need to restructure the loop so that (1) the iteration count of the loop is a multiple of k and (2) that there are k exposed values to operate on. Similarly we can separate the expected hot path in the loop body from the branches that may lead to abnormal loop exits. This restructuring can be viewed as a variation on the *hot-trace* with a *guarded trace-exit* flow restructuring that is commonly done in *Tracing Just-In-Time Compilers* [1, 7].

2.2 Relational Verification of SIMD Loops

In this work we are primarily interested in showing the equivalence of a scalar loop and a loop using SIMD instructions. Thus, to leverage the relational verification machinery we need to identify a suitable set of equivalence relations that may hold between a scalar loop and the corresponding SIMD implementation. We have identified two commonly occurring forms for these equivalence relations, invariant and reduction expressions Section 4, which are sufficient to enable the scalar/SIMD loop equivalence verifications we are interested in. Consider the following loops which illustrate the needed equivalence relations:

```
int x = ...;
int hash, i = 0;


for(; i < n; i += 4) {
  hash ^= A[i] & x;
  hash ^= A[i+1] & x;
  hash ^= A[i+2] & x;
  hash ^= A[i+3] & x;
}
```

```
int x = ...;
int hash, i = 0;
m128i hv = [0, 0, 0, 0];
m128i xv = [x, x, x, x];
for(; i < n; i += 4) {
  m128i d = load_128(A + i);
  m128i t = and_i128(d, xv);
  hv = xor_i32(hv, t);
}
hash = (hv.r0 ^ hv.r1 ^
        hv.r2 ^ hv.r3);
```

The first loop on the left contains several variables with live ranges that span multiple iterations of the loop. The variable `x` is an invariant value in the loop on the left and a vectorized invariant version `xv` is used in the second loop on the right. The variable `hash` is a reduction variable in the first loop. The loop on the right represents these accumulated values in four `i32` values in the vector variable `hv` and adds a final reduction at the exit of the loop. Thus, the relations needed to show these loops are equivalent on each lockstep iteration are: $i_{\langle 1 \rangle} = i_{\langle 2 \rangle} \wedge xv = [x_{\langle 1 \rangle}, x_{\langle 1 \rangle}, x_{\langle 1 \rangle}, x_{\langle 1 \rangle}] \wedge hash_{\langle 1 \rangle} = (hv.r0 \hat{} hv.r1 \hat{} hv.r2 \hat{} hv.r3)$.

Given these relations it is straight forward to use a relational verification technique to show the program fragments are equivalent. In practice we use a *product program construction* [4] with off the shelf SMT solvers to solve the generated verification conditions. The equivalence relation is clearly satisfied on the first entry to the loop. Then inductively we can see that if the equivalence holds on iteration k then in iteration $k+1$ it will again hold after executing both loop bodies. The final step is then simply to show that when the loop exits, and after executing the final reduction after the loop when the relational equivalence invariant holds, that $hash_{\langle 1 \rangle} = hash_{\langle 2 \rangle}$.

In practice these new relations, *invariant expression vectorization* and *reduction variable vectorization*, along with previously known relations (Section 2.1) for reasoning about loop control-flow restructuring are sufficient to verify the equivalence of the loops that are of interest in this work. In Section 4 we will formalize the definitions for the invariant and reduction vectorization equivalence relations. Additionally, we show how to leverage the relational verification methodlogy to construct the constraints needed to synthesize a vectorized body given a scalar implemention of a loop.

3. Problem Description & Algorithm Overview

This section presents a formalization of the program fragment language that we want to vectorize and the language with SIMD instructions that the auto-vectorization algorithm produces as output. We also provide an overview of the auto-vectorization algorithm.

3.1 Input and Output Loop Languages

Input Language. The work in this paper operates on a core imperative language shown in the non-bold portion of Figure 3. For simplicity, this language consists of variables and operations on three types: 32-bit integers, user defined structures (with named fields) and arrays of either structures or integers. This language extends naturally to include other integer sizes, floating point values, etc. The expressions e in the language cover the standard sets of arithmetic, bitwise, comparison, and access operations. The language admits the standard suite of assignments to locals, array locations, and fields in structs.

To focus on blocks of code that are suitable for vectorization, we distinguish between blocks consisting of simple assignments with conditional flow inside a single iteration b, and blocks of statements that may contain non-local control flow f. The grammar describes the structure of the loops that we are interested in vectorizing and which are likely to benefit most from the conversion to a SIMD implementation – innermost loops that are free of function calls. However, in practice the technique can be applied more aggressively by explicit inlining of function calls or by providing explicit pre/post semantics for an inner loop or method call.

To ensure that the loop is amenable to vectorization, we also impose some semantic restrictions: (1) the loop limit expression e' is invariant, (2) the iteration variable is only updated by linear operations, and (3) the updates are done uniformly on all paths of the loop. Finally, we define a program fragment δ as a single loop with possible loop initialization and clean-up code.

SIMD Output Language. The output of the auto-vectorization algorithm is a program in the SIMD extended language shown in Figure 3, including the terms in bold. The output language extends the input language with a set of SIMD instructions similar to what is present in the Intel SSE4 instruction set. For simplicity, we assume that all of the vectors ($v \in V$) are 128 bits which can contain 4 integers of 32 bits each. We extend the constant set with macros for shuffle constants and add the `allzeros` operation to the set of expressions that produce integer values (e). The SIMD expressions (**ve**) treat each 128 bit vector either as a single bit set of 128 bits for logical operations (*e.g.* `or_i128` or `and_i128`) and as four 32 bit integer values for arithmetic and comparison

operations (*e.g.* `add_i32` or `cmpgt_i32`). We add operations to load (`load_128`) and store (`store_128`) 128 bits at a time. Finally, we allow the fragment to contain a sequence of loops.

3.2 Algorithm Overview

The auto-vectorization algorithm is depicted in Figure 4. This flow diagram shows how we first apply deductive restructuring to the loop to expose data parallelism using deductive rewritings. From this restructured loop and the associated equivalence relations from Section 2 we extract a loop-free block of code from the loop body which will be replaced with a sequence of synthesized SIMD instructions. This synthesized code is then patched back into the loop. Finally we compute a cost scoring function and a proof of correctness for the final code fragment.

Restructuring and Pre/Post Generation. The loop is first restructured via standard loop splitting/unrolling and if-conversion. We also introduce vector variables (Section 4) which are used in the synthesis phase. The condition generator examines the restructured program and equivalence relations that are built up during the resturcturing to construct the needed synthesis pre/post conditions.

Inductive Concolic Synthesis. The synthesis phase (Section 5) takes the pre/post conditions produced by the previous step and produces a sequence of instructions that realize the specified behavior. The synthesizer uses a novel combination of concrete program execution and counter example generation, which we call *concolic synthesis*. This combined search approach quickly produces an efficient sequence of instructions that satisfies the pre/post conditions and this sequence of instructions is the output of this phase.

Merge and Cost Ranking Function. The final step in the algorithm is to patch in the synthesized code for the hole in the program and to clean up any dead or loop invariant code that may have been created in the vectorization step. The final program is passed to the cost ranking computation (Section 6), and a proof of correctness is computed for the program.

Output. The output of the algorithm is (1) the SIMD optimized program, (2) a proof of equivalence between the SIMD implementation and the original program, and (3) a cost ranking function.

This approach provides the ability to use deductive heuristic rules to quickly rewrite a loop to expose parallelism and enables the reduction of the synthesis problem to small blocks of code. The synthesis component provides a simple inductive method to construct efficient code blocks that is robust to a wide range of structures, and variations on these structures, that appear in the loop bodies. The relational verification methodology provides a connection between the inductive and deductive approach allowing them to co-operate to maximize the strengths of each approach. As an additional benefit the correctness certificate enables the pre-compilation of code in a managed language, such as C#, to assembly code which can be deployed and JITed without violating the safety guarantees of the language [26, 27].

4. From Relational Verification to Synthesis

We borrow the general concept of turning an appropriate verification methodology into a synthesis algorithm from [38] and extend the core idea to apply it to our problem domain. The approach in [38] requires a specification of the program to be synthesized as a pre/post condition pair (ϕ, ψ) and a template T (with only first-order holes) that form the full-correctness proof for the program. In our setting there are two natural possibilities for constructing the pre/post conditions: (1) the minimal loop invariant required for correctness and (2) the precondition and postcondition for the loop body. Unfortunately, both of these options are unsatisfactory. The computation of loop invariants (even with the limited language in Figure 3) is an undecidable problem. Conversely, pre/post conditions based on only the loop body can be computed efficiently. However, the resulting conditions are highly restrictive and cannot be used for loops that require certain vector registers to be live across loop iterations (such as reduction variables).

We use results from the area of *Relational Program Verification* [4, 8, 30] and the transformation/verification rules outlined in Section 2 to generate the synthesis pre/post conditions. To expose or create data parallelism which can be exploited in the SIMD synthesis step we utilize standard loop restructuring rules (splitting, unrolling, and if-conversion) and rules for introducing vectorized variables or constants. Each rule consists of (1) a loop rewriting template and (2) a template for the equivalence relation between the original loop and the rewritten version. The equivalence relations are used to generate the desired synthesis condition and an equivalence proof between the original loop and the vectorized version (or to reject the vectorized version if a proof cannot be generated).

4.1 Introduction of Vectorized Variables/Constants

Vectorization of Loop Invariant Expressions. We identify variables and expressions e that are invariant across loop iterations in the standard manner – either none of the values used in e are modified in the loop body or they are assigned the result of another loop invariant expression. For each invariant expression, e of type Int32, we introduce the corresponding *vectorized* version, $\texttt{vece} := \langle e, e, e, e\rangle$, where `vece` is a fresh variable name and the initialization is done before the loop. We accumulate the loop invariant expressions and the corresponding vector variables as tuples $(e, \texttt{vece})$ in the set V_e. The equality relationship that should exist between the scalar and vector forms is given by:

$$\mathsf{Inv}(V_e) = \bigwedge_{(e,v)\in V_e} v = [e_{\langle 1\rangle}, e_{\langle 1\rangle}, e_{\langle 1\rangle}, e_{\langle 1\rangle}]$$

Vectorization of Reduction Variables. We define a reduction variable x as a candidate for reduction variable vectorization when: x is not used as an array index and all paths through the loop contain an assignment of the form $x := x \bullet e$ where $\bullet$ is commutative. This definition heuristically identifies a reduction variable x, introduces a vector version `vecx`, and adds the appropriate initialization before the loop with reduction at loop exit. This definition is unsafe to use in general as we have ignored the effects of the rest of the loop body and their interaction with the reduction operator. However, in the case where the transformation is unsafe we will not be able to produce a proof of equivalence in the relational verification step and will reject the resulting program. The equality relationship that should exist between the scalar and vector forms is given by:

$$\mathsf{Reduce}(V_r) = \bigwedge_{(x,v,\bullet)\in V_r} \begin{pmatrix} v = [r_0, r_1, r_2, r_3] \wedge \\ x_{\langle 1\rangle} = x_{\langle 2\rangle} \bullet (r_0 \bullet r_1 \bullet r_2 \bullet r_3) \end{pmatrix}$$

4.2 Final Equivalence Relation

Once we have the identified the loop invariant expressions, the reduction variables, and the input and output variables (I and O) for the loop, the next step is to construct the final equivalence relations at the desired synchronization points.

In our setting the synchronization points correspond to the program points at the normal control-flow entries and exits of the loop bodies. By definition the variables in V_e or V_r are in scope at both the loop body entry and exit points so the special equality conditions for them are the same at both points. For the variables not in the V_e or V_r sets we check if they are in the input or output variable sets (I or O) and if so add an equality condition between the versions in the two programs. We say a variable x is *simple* at a

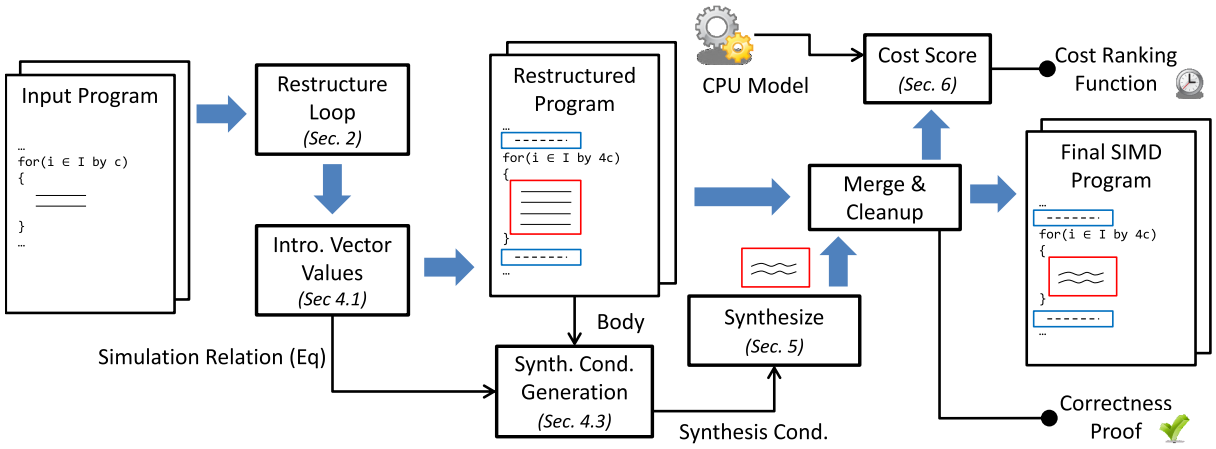

Figure 4. Overview of the auto-vectorization algorithm.

program point if it is defined at that point and it is not an invariant or reduction variable. We define the equivalence relation for the loop entries E_{pre} and exits E_{post} as:

$$E_{pre} = \mathsf{Inv}(V_e) \wedge \mathsf{Reduce}(V_r) \wedge \bigwedge_{x \in X} x_{\langle 1 \rangle} = x_{\langle 2 \rangle}$$
$$\text{where } X = \{x \in I | x \text{ simple at the loop entries}\}$$
$$E_{post} = \mathsf{Inv}(V_e) \wedge \mathsf{Reduce}(V_r) \wedge \bigwedge_{x \in X} x_{\langle 1 \rangle} = x_{\langle 2 \rangle}$$
$$\text{where } X = \{x \in I \cup O | x \text{ simple at the loop exits}\}$$

4.3 Partial Program and Condition Generation

As the natural candidate code for conversion to a SIMD implementation is the normal control-flow block of the loop body. We replace the normal control-flow block between the loop entry and exit with a *hole* [36]. Using the equality relations from E_{pre} and E_{post}, along with the *weakest preconditions* computed with them, we can construct pre/post conditions ϕ and ψ for the hole which are used to construct replacement code to fill the hole.

Given our choice synchronization points as the loop normal control-flow entries/exits, the required verification condition is of the form $E_{pre} \Rightarrow \mathsf{wp}(b_1, \mathsf{wp}(b_2, E_{post}))$ where b_1, b_2 are the loop bodies from the left and right programs respectively and wp computes weakest preconditions. If b_2 is a *hole* then this verification condition is a specification for the required code. We compute the synthesis pre/post conditions (ϕ, ψ) for our synthesis hole by taking the code for the block we want to replace and compute:

$$\phi = \exists I_{\langle 1 \rangle} E_{pre} \text{ where } I_{\langle 1 \rangle} = \{x_{\langle 1 \rangle} | x \in I\}$$
$$\psi = \exists I_{\langle 1 \rangle} (E_{pre} \wedge \mathsf{wp}(b_1, E_{post})) \text{ where } I_{\langle 1 \rangle} = \{x_{\langle 1 \rangle} | x \in I\}$$

This construction lifts the relational program verification methodology to a synthesis condition generation methodology. Further, it reduces the problem of synthesizing loopy programs to the problem of synthesizing straight line code. However, it does so in a way that preserves cross loop information as well as context from before/after the loop body. This context ensures that the generated conditions are as *relaxed* as possible, enabling the generation of optimized code in the synthesizer, while still ensuring the equivalence of the original and optimized programs.

4.4 Running Example

Figure 5 shows the result of applying these transformations to the input code from Figure 1. The resulting fragment has two loops, the first one has a loop guard that ensures the loop iteration count is a multiple of 4 while the second loop handles the remaining iterations. The first loop has been unrolled 4 times and the variables have been uniquely renamed to expose 4 independent sets of values for the vectorization. The if-conversion step has swept the conditional guards and abnormal loop exit to the single flow block at the end of the loop.

After the loop restructuring and introduction of vector variables, `vectv` and `vecsv` for `tv` and `sv` respectively, we have the following following equivalence post relation for the body:

$$\begin{aligned} E_{post} = \; & match_{\langle 1 \rangle} = match_{\langle 2 \rangle} \wedge vals_{\langle 1 \rangle} = vals_{\langle 2 \rangle} \\ & \wedge i_{\langle 1 \rangle} = i_{\langle 2 \rangle} \wedge tv_{\langle 1 \rangle} = tv_{\langle 2 \rangle} \wedge sv_{\langle 1 \rangle} = sv_{\langle 2 \rangle} \\ & \wedge vectv = [tv_{\langle 1 \rangle}, tv_{\langle 1 \rangle}, tv_{\langle 1 \rangle}, tv_{\langle 1 \rangle}] \\ & \wedge vecsv = [sv_{\langle 1 \rangle}, sv_{\langle 1 \rangle}, sv_{\langle 1 \rangle}, sv_{\langle 1 \rangle}] \end{aligned}$$

The code shown in Figure 6 has had the normal control flow code in the loop, from the first statement to the `if`, replaced with a `[HOLE]` as a place holder for the code we want to synthesize. The pre/post conditions we want to generate (ϕ and ψ) for use in the synthesis step are shown before/after the hole.

The only assignments to externally visible variables that can be made by the synthesized code are specified by the set O. Thus, we simplify the computed post condition, ψ, by assuming that all variables not in O have the same values before/after the synthesized code. As the only variable in O is `match`, the interesting parts of the generated synthesis pre/post conditions are:

$$\begin{aligned} \phi = \; & (vectv = \langle tv, tv, tv, tv \rangle \wedge vecsv = \langle sv, sv, sv, sv \rangle) \\ \psi = \; & (match = ((vals[i].tag = tv \wedge vals[i].score > sv) \\ & \vee (vals[i+1].tag = tv \wedge vals[i+1].score > sv) \\ & \vee (vals[i+2].tag = tv \wedge vals[i+2].score > sv) \\ & \vee (vals[i+3].tag = tv \wedge vals[i+3].score > sv))) \end{aligned}$$

After synthesizing the SIMD code for these conditions and substituting it in for the hole we get the final program shown in Figure 2. Using the equivalence relations E_{pre} and E_{post} we can compute and discharge a set of verification conditions for the original input loop and the final SIMD implementation which serve as a correctness proof for the transformation. Finally, using the construction in Section 6 we can produce a cost function for the relative performance of the input and SIMD loops.

```
int i;
for (i = 0; i < len-3; i+=4) {
  int tagok0 = vals[i].tag == tv;
  int scoreok0 = vals[i].score > sv;
  int andok0 = tagok0 & scoreok0;

  ...

  int tagok3 = vals[i+3].tag == tv;
  int scoreok3 = vals[i+3].score > sv;
  int andok3 = tagok3 & scoreok3;

  match = andok0 | andok1 | andok2 | andok3;
  if (match) return 1;
}

for (; i < len; ++i) {
  int tagok = vals[i].tag == tv;
  int scoreok = vals[i].score > sv;
  int andok = tagok & scoreok;
  if (andok) return 1;
}
```

Figure 5. Running example after structural transformation.

```
int i;
for (i = 0; i < len-3; i+=4) {
  φ
  [HOLE]
  ψ
  if (match) return 1;
}

for (; i < len; ++i) {
  int tagok = vals[i].tag == tv;
  int scoreok = vals[i].score > sv;
  int andok = tagok & scoreok;
  if (andok) return 1;
}
```

Figure 6. Running example after hole insertion and pre/post condition locations shown.

5. Inductive SIMD Synthesis

The synthesis algorithm takes a pre/post condition pair (ϕ, ψ), a set of instructions to select from *Stmts*, the set of input variables I and outputs O, and a maximum cost for the program to be synthesized ($cost_m$). The output is a program p which is a sequence of statements such that for any state valuation s that satisfies the precondition ϕ, the execution of p starting in s yields a state valuation s' that satisfies the postcondition ψ. Inspired by work on *concolic testing* [9, 32] our concolic synthesis algorithm uses a combination of a top-level counter-example driven loop (based on symbolic methods) to find interesting values for the inputs I and an efficient search for candidate programs p (based on concrete execution over these input values). The symbolic reasoning in the top-level loop (Algorithm 1) ensures that each new input provides useful information, which forces behavioral differences, while the use of concrete values in the program search subroutine (Algorithm 2) provides an efficient method for generating candidate programs.

5.1 Counter-Example Generation Loop

The top-level *CEGIS* (Counter-Example Guided Inductive Synthesis [35]) loop in Algorithm 1 iteratively constructs a set of *concrete state valuations* (a mapping of values to variables) and searches for a candidate program p that satisfies the postcondition ψ when run on these state valuations, line 6. On line 4 the algorithm attempts to symbolically construct a new input state valuation s that is a counter-example for the correctness of the program p – *i.e.* ψ does not hold on the result of running p on s. If such an example can be found it is added to the set on line 5 and the loop is repeated, if we can prove that no such example exists then p is the desired program and we return on line 8, and if we cannot decide if such an example exists then the synthesis fails. The initialization of the concrete state valuations set, the underlined call to *GenInitialStates* on line 2 is an optimization, described in Section 5.3, to minimize the number of iterations of the CEGIS loop.

Algorithm 1: Top-Level CEGIS Loop

input : pre ϕ, post ψ, statements *Stmts*, inputs I, outputs O, max. cost $cost_m$, disjunctive precondition χ
output: program p

1 $p \leftarrow$ skip;
2 $S \leftarrow \underline{\mathit{GenInitialStates}(\chi)}$;
3 **while** GenModel$(\exists \vec{V}, \phi \wedge \neg \text{wp}(p, \psi)) \notin \{\text{unsat}, \text{fail}\}$ **do**
4 $\quad s \leftarrow \mathit{GenModel}(\exists \vec{V}, \phi \wedge \neg \mathit{wp}(p, \psi))$;
5 $\quad S \leftarrow S + s$;
6 $\quad p \leftarrow \mathit{Search}(\langle\rangle, S, \psi, \emptyset, \emptyset, \mathit{Stmts}, I, O, \mathit{cost}_m)$;
7 $\quad$ **if** $p = \perp$ **then return** fail;
8 **return** (GenModel$(\exists \vec{V}, \phi \wedge \neg \text{wp}(p, \psi)) = \text{unsat}$) *?* p *: fail*;

5.2 Candidate Program Search

The *Search* method, Algorithm 2, performs the search for a program p_{res} that when run on the input list of state valuations S produces a list of state valuations that satisfy the post condition ψ. The *naive search*, i.e., the algorithm excluding the underlined code, is a depth first enumeration of possible sequences of instructions from the set *Stmts*. If we reach a point where every state valuation in S satisfies ψ then we have a candidate program and can return it, line 9. Otherwise the current program is extended with another instruction from *Stmts*, yielding p_i, and this statement is applied to each of the state valuations in S, yielding S_i. The new values, p_i and S_i, are then used in the recursive search call on line 14. As this naive search approach is computationally intractable for instruction sequences of length greater than four [12, 16, 23] we introduce several optimizations below.

5.3 Synthesis Optimizations

Initial State Valuations. The set of input state valuations S in the top-level CEGIS loop (Algorithm 1) plays a critical role in the number of iterations required for the loop to terminate. Every new state valuation that is added to S is, by construction, a counter-example and when no further counter-examples can be generated the loop terminates. Thus, we initialize S with a number of input valuations that are likely to provide good initial constraints and as a result we will need to generate very few additional counter-examples. As in concolic testing we note that different paths through the program are likely to exercise different behaviors. Thus, we alter the synthesis algorithm to take a *disjunctive pre-condition* χ, which is a disjunction of *per path* weakest preconditions from the input program. The *GenInitialStates* method produces a state valuation for each clause in the disjunctive pre-condition and we use these to initialize S on line 2.

Search Merging. The naive search builds redundant instruction sequences that repeatedly generate the same program state valuations, *e.g.* repeatedly add and then subtract a constant. We also observe that the search re-explores equivalent state valuations that are reachable on different instruction paths, *e.g.* $(a + (b + c)) - d$ and $(a + b) + (c - d)$. We can eliminate this redundant exploration by merging branches in the instruction sequence search tree that are actually exploring the same set of state valuations. This is

done by adding a set, *Seen*, of state valuations that have been seen during previous search steps (checked on line 2). If we encounter a state valuation that has been previously seen it means that either (1) the current instruction sequence has redundant instructions, in which case it is suboptimal, or (2) we have already explored the state valuations reachable from the current valuation, and so continuing exploration on this sequence of instructions merely re-visits previously seen state valuations. In these cases, pending a check on cost information described below, we simply abort the current branch of the search on line 3.

Cost Bounds. In our application we are only interested in minimal cost code sequences. Using the cost model from Section 6 we score the initial program fragment that is being replaced and compute cost scores for each program generated during the search. With this information we can immediately stop searching, line 1, if the current program has a larger score, $cost_m$, than the input program or current best solution. This bound is updated as needed to be the best found so far in a standard branch-and-bound manner, line 17.

We further refine how the search handles state valuations that have been seen previously by noting that computational cost is monotone. Thus, repeating the exploration of a previously visited state with a higher cost program will not discover a faster program. However, if the current instruction sequence was able to produce the current state valuation more efficiently than previous instruction sequences then it may be possible to reach target state valuations satisfying ψ without exceeding the cost bound $cost_m$. Thus, on line 3 we check if we have found a less costly instruction sequence, if not we return immediately but if the new instruction sequence is less costly we update the min cost for this state valuation on line 4 and continue the search (re-exploring as needed).

Stack Machine. In order to limit the introduction and lifetime of intermediate values, as well as to reduce the combinatorial problems of selecting which variables to use/modify in each instruction, we extend the concrete execution state valuation with an evaluation stack. The use of an evaluation stack is a common way to simplify the operation of an abstract machine (*e.g.* the .Net and Java virtual machines) by removing the need to explicitly refer registers or to introduce explicit temporary variables. In the instruction selection step, line 11, we assume instructions take their arguments from the evaluation stack and place the result on the stack. We also extend the instruction set with operations to load input variable values on the stack and to pop values off of the stack into output variables. The introduction of an explicit evaluation stack allows us to place a bound on stack depth, line 8. This biases the search to avoid instruction sequences that produce large numbers of intermediate values which would produce code with high register pressure.

Incremental Search Expansion. We can obtain additional performance by using incremental expansion of the search parameters. In general the operations used by the original program (`==`, `&`, ...) are the same type of operations that will be needed in the SIMD version. Thus, we start with only the corresponding vector operations and basic load/store operations in the set of instructions (*Stmts*). If we fail to find a suitable program we extend this set with additional operations such as the `shuffle` and other bitmasking operations. Finally, if this larger set fails we let *Stmts* be the set of all instructions. Similarly, we start with a small eval stack, in our case depth 4 increasing to 6 if the first search fails. This allows us to improve the performance of the synthesizer in many cases but still allows the incremental exploration of the full program space as desired.

Algorithm 2: Concrete Program State Search

input : program p, state valuations S, post ψ,
seen set *Seen*, seen cost *Cost*, instructions *Stmts*,
inputs I, outputs O, max. cost $cost_m$
output: program p_{cand}

1 **if** $\text{cost}(p) \geq cost_m$ **then return** $\perp$;
2 **if** $S \in Seen$ **then**
3 **if** $cost(p) \geq Cost(S)$ **then return** $\perp$;
4 $Cost \leftarrow Cost + [S \rightarrow cost(p)]$;
5 **else**
6 $Seen \leftarrow Seen \cup \{S\}$;
7 $Cost \leftarrow Cost + [S \rightarrow cost(p)]$;
8 **if** $\text{Stack}_{\text{depth}}(S) > \text{Max}_{\text{stack}}$ **then return** $\perp$;
9 **if** $\forall s \in S \,.\, \psi$ *holds for* s **then return** p;
10 $p_{res} \leftarrow \perp$;
11 **foreach** stmt $\in$ Stmts$\cup\{\text{ldv}(v)|v \in I\} \cup \{\text{stv}(v)|v \in O\}$ **do**
12 $p_i \leftarrow p + inst$;
13 $S_i \leftarrow ApplyInstToAll(inst, S)$;
14 $p_o \leftarrow Search(p_i, S_i, \psi, Seen, Cost, Stmts, I, O, cost_m)$;
15 **if** $p_o \neq \perp \wedge cost(p_o) < cost_m$ **then**
16 $p_{res} \leftarrow p_o$;
17 $cost_m \leftarrow cost(p_o)$;
18 **return** p_{res};

6. Cost Ranking Function

The computation of absolute costs for arbitrary blocks of code is a challenging problem [39]. However, we do not need to compute the absolute costs of the programs. As we are only interested in identifying the best performing program from a set of candidates we only need to model cost in a way that allows relative comparison of two programs. Further, our more restricted program fragment language and vectorization application possess a number of simplifying features. The impacts of branch mis-prediction are parameterized as described below while the the uniform array accesses required for vectorization imply that the caching/prefetching in the processor will behave in a consistent and uniform manner.

We assume that we are given a model of the processor architecture, M, which contains the standard information on execution unit resources and latencies as well as branch mis-predict costs M_{miss}. We parametrize the remaining program fragment behaviors based on the conditionals C (*i.e.* `if` statements) and the loops L that appear in the program fragment:

$B_p : C \mapsto [0, 1)$	The mis-predict probability of each branch.
$B_t : C \mapsto [0, 1)$	The probability that the true path is taken.
$L_c : L \mapsto \mathbb{N}$	The number of times a loop is executed.

From these parameters we construct a cost ranking function $Perf^M : (\delta, B_p, B_t, L_c) \mapsto \mathbb{R}$. The cost of a straight line block of code is simply the sum of each statement as reported by the underlying processor model M. The cost of a branch statement, β with true branch β_t and false branch β_f is:

$$\begin{aligned} &Perf^M(\beta, B_p, B_t, L_c) = \\ &\quad B_p(\beta) * M_{\text{miss}} + B_t(\beta) * Perf^M(\beta_t, B_p, B_t, L_c) \\ &\quad + (1 - B_t(\beta)) * Perf^M(\beta_f, B_p, B_t, L_c) \end{aligned}$$

The cost of a loop statement, ℓ with the body ℓ_{body} is simply $M_{\text{miss}} + L_c(\ell) * Perf^M(\ell_{body}, B_p, B_t, L_c)$. We can compute the cost ranking function for a fragment where $\delta = f_{init}; \ell_1 \ldots \ell_k; f_{exit}$ in the natural way as the sum of all the costs:

$$\begin{aligned} &Perf^M(\delta, B_p, B_t, L_c) = Perf^M(f_{init}, B_p, B_t, L_c) \\ &\quad + (\Sigma_{\ell_1 \ldots \ell_k} Perf^M(\ell_i, B_p, B_t, L_c)) + Perf^M(f_{\text{exit}}, B_p, B_t, L_c) \end{aligned}$$

In this paper we report results when running the code on an Intel i7 processor. We can construct a (very) simple model M for this processor with: a normalized latency of 1 per operation, a 3 wide execution unit, a mis-predict cost $M_{\text{mis}} = 12$, a uniform mis-predict probability of 5% for forward conditional branches, and a 1% mis-predict rate for loop back and exit branches. Using this model the cost ranking function for the SIMD loop in Figure 2 is $Perf_{\delta}^{M}(\delta_{opt}, n_1, n_2) = 24 + n_1 * 5.16 + n_2 * 3.16$ and for the original loop in Figure 1 the function is $Perf_{\delta}^{M}(\delta_{orig}, n_1') = 12 + n_1' * 3.16$.

The estimated asymptotic speedup can be computed by observing that as n_1' becomes large the costs of the loops are proportional to $n_1' * 3.16$ for the original loop and $n_1' * 1.29$ for the SIMD loop (since the SIMD loop processes 4 elements per iteration). Thus, the cost ranking functions predict a speedup of $2.44\times$, closely matching the empirically observed speedup of $2.5\times$. To find the predicted break even point we solve for the values where the cost ranking functions for the original loop and SIMD loop are equal, $n_1' = 8$ for our functions. As we see in Section 7 this matches well with the experimentally seen break-even of between 4 and 8.

Even with our simple processor model, which can be built using readily available information, the resulting static cost predictions are both precise and, as we would like in a static compiler, conservative. This simple model can be further improved via either more detailed architecture descriptions [39] or autotuning [41] to identify key performance parameters in the processor models. As the cost estimation functions are parametrized on branch mispredict, branch taken, and loop count information it is also possible to evaluate them and select the best implementation based on runtime data, as in a Tracing JIT [1, 7].

7. Experimental Evaluation

To evaluate the approach presented in this paper we selected 18 loops which represent fundamental classes of algorithms (find, exists, accumulate, map, etc.) that are found in standard libraries such as the STL for C++ or the base class libraries for C# (or Java). These algorithms cover many common loop idioms that appear in real world code. In this section we examine 6 benchmarks in detail. Four benchmarks – CountIf, Find, Lexo, Equals – come from the C++ STL (specialized for random access iterators). Two benchmarks – FindIf and the running example Exists – are from the .Net base class libraries (BCL) and are implementations of methods in the `List<T>` class. Finally, the CyclicHash comes from production C++ code and implements a hash code function for data blocks. For the methods that take user defined lambda expressions – CountIf, FindIf, and Exists – we used non-trivial instantiations for the lambda code, *e.g.* the running example in Figure 1.

7.1 Transformation and Synthesis Performance

Table 1 shows the time required to vectorize each program fragment (memory is always less than 100MB). In practice the synthesis step accounts for 90% or more of this time. Thus, the table shows the number of input states the synthesis started with, the number of additional iterations the CEGIS loop needed, and the number of instructions in the final synthesized block. As we can see in this table the resource requirements vary greatly even for similarly sized code blocks. This variability is not surprising as the synthesis is fundamentally a search in a very large state space. However, in all of the cases the synthesizer was able to produce an optimized SIMD program. These programs consisted of up to 9 instructions and covered a diverse set of comparison, bitwise, and swizzling operations. The results also show the impact the *disjunctive precondition* generation heuristic has on the total number of iterations (taking only 1 iteration for all but one case).

Benchmark	Time(s)	Init./Iters.	Insts.
CountIf	0.136s	16/1	8
Find	0.053s	6/1	4
Lexo	0.056s	6/1	5
Equals	0.667s	10/1	5
Exists	0.120s	6/2	9
CyclicHash	0.998s	16/1	5

Table 1. *Time* required by the synthesizer. *Init* number of examples in S and *Iters* of the CEGIS loop. *Insts* in the final SIMD code.

7.2 Performance of SIMD Loops

To compare the performance of the synthesized SIMD loops and the original scalar implementations we implemented a driver loop which executes each loop, on inputs of various sizes, 5 million times in a simple timing loop. The evaluation was done on an Intel i7 running Windows 7 (32 bit) and Visual Studio C++ compiler (Version 16 for x86) with the default optimization settings.

Figure 7 contains a chart for each of the benchmark loops. This chart shows the experimentally measured performance improvement seen with the synthesized SIMD implementation and the performance improvements predicted by the analytical cost functions in Section 6. The logarithmic x-axis is the number of iterations that the original loop expected to execute. For fixed count loops like CountIf and CyclicHash this is the size of the input array. For loops with abnormal returns (the remaining four loops) this is the expected number of iterations before the loop exits. As we use a uniform distribution for where the element of interest is in the input the expected number of iterations is half the length of the input array. The y-axis is the speedup of the SSE implementation relative to the original scalar implementation. Finally we mark the break-even line where the performance of the SSE implementation and original implementation are equal.

The results in Figure 7 show that in general the SIMD implementations start to outperform the baseline implementations almost immediately (the *Actual* plot). For an iteration count of 8 only the Lexo loop is slower than the baseline implementation while CyclicHash is slightly better than break-even and the remaining loops show a 10% to 40% reduction in runtime. As the input size gets larger the performance differences get larger in favor of the SIMD loops. Once the iteration counts approach 32 the Lexo has passed the predicted break even point and is now faster than the baseline implementation. At iteration counts of 512 all the loops outperform the baseline by a factor of $2\times$ or more. Finally by iteration counts of 2048 the loops performance ratios are near their asymptotic speedup and now outperform the baseline implementations by between $2.0\times$ and $3.7\times$, which is near the $4\times$ maximum speedup we would expect from using 4 wide SSE instructions. These results demonstrate that the approach to vectorization described in this paper is *applicable* to a wide range of loops and produces SIMD implementations that consistently provide large performance increases (even on relatively small inputs).

The *Predicted* plots in Figure 7 show that in general the speedups predicted by the analytic cost model from Section 6 correlate well with the observed speedups – despite the relatively crude model used for the processor. The major exception to this trend is the Equals program where the predicted and actual performance diverge significantly for large iteration counts. Further investigation indicates that in this case the processor is able to optimize the loop execution in ways that are not captured by the simple processor model, M, used when constructing the cost functions. Thus there is room for improvement via either more detailed architecture descriptions [39] or autotuning [41] to identify key performance parameters in the processor models.

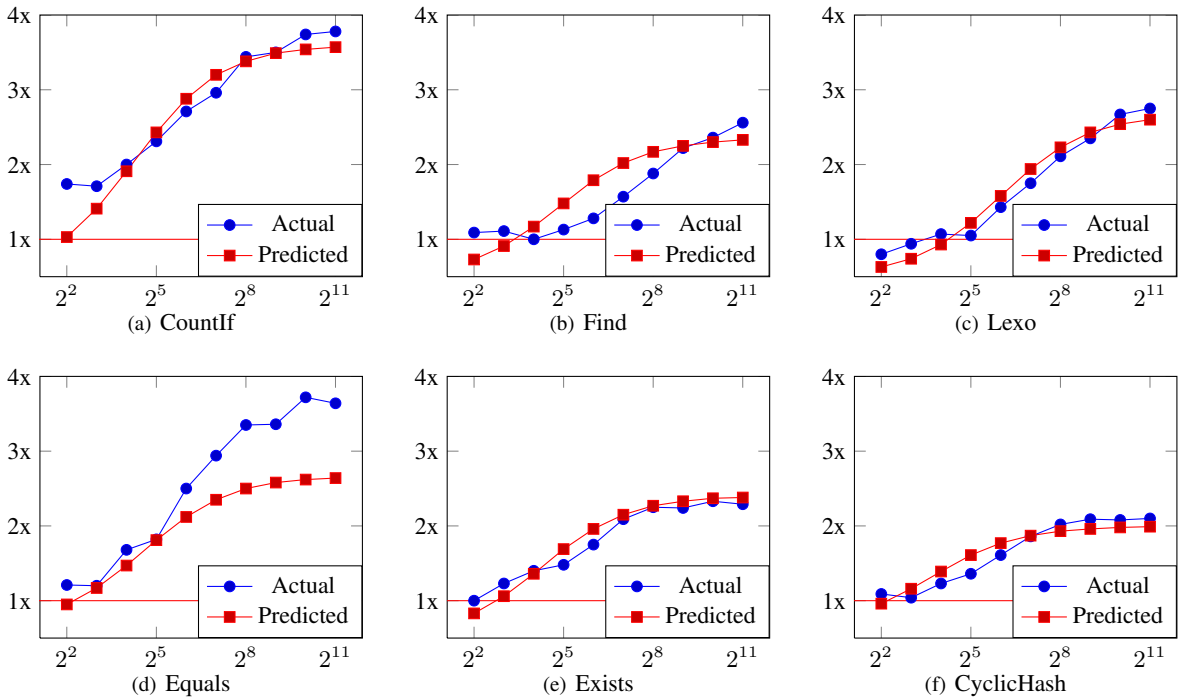

Figure 7. Speedup (Original Time / SSE Time) on Y axis ranging over the expected number of iterations in original loop on X axis.

To avoid performance degradations it is critical that the cost model is able to predict the break-even number of iterations (where the SIMD loop begins to outperform the original loop). In all our benchmarks we see that this number is well predicted by the cost model and in all cases is a conservative estimate (*i.e.* overestimating the number of iterations needed to break-even). Thus, these results demonstrate that the cost model defined in this work is an effective predictor for the relative performance of the loops and provides an effective means to check that a SIMD implementation will *reliably improve* the performance of the program in practice.

To validate that these results were not an artifact of the evaluation environment [25] we ran the evaluation on a second platform. This environment consisted of an Intel Core2 running Mac OS X (Tiger) and GNU C++ compiler (Version 4.2.1 x86). The results were, with one notable exception, consistent with the performance improvements seen on the Intel i7 platform. The outlier benchmark, *Exists*, had a break-even cost of 16 on the Core2 compared to a break-even of 8 on the i7. This increase is mainly a result of the `shuffle` operation being more expensive on the Core2 and the increased break-even is correctly predicted by our cost function.

7.3 Synthesis with Specialized Operations

In order to evaluate how the synthesis technique handles SIMD operations with unusual semantics we synthesized three common string operations from the C# `System.String` class: StringEquals, IndexOf and IndexOfAny. These can be implemented using the specialized *Packed Compare Strings* (PCMPESTRI) operation from SSE 4.2. The synthesis algorithm produces SIMD implementations for these loops using the specialized packed compare strings operation in less than 1 second for each benchmark. The speedups obtained ranged from 3.4× for StringEquals to 9.5× for IndexOfAny. These results demonstrate how the synthesis approach can be easily extended to make use of new, or unusual, instructions to produce optimized loop implementations.

7.4 Impact on 483.Xalan

The results in Section 7.1 show that the synthesized loop implementations consistently improve performance across a range of loops and input data sizes. To validate that the performance gains seen on the micro-benchmarks translate into similar performance gains in practice, we selected the 483.Xalan benchmark from SPEC CPU2006 [37] as a case study. This program makes heavy use of `std::vector<string*>` as a cache for commonly used strings and it uses the STL `find` algorithm (our Find benchmark) to find string pointers in the cache.

The cache behavior is very sensitive to the data that is being processed as shown in recent work on automatic data structure selection [17]. Replacing the `std::vector` with a `std::set` (or a `hashset`) resulted in performance improvements of up to 20% on the SPEC provided `train` input but when run with the SPEC provided `test` input the alternative data structure representation actually *degrades* performance by up to 20%. This swing from performance improvement to performance degradation is driven by the sensitivity of the cache to particular features of the input data set. Thus, this program tests both the performance impact of the SIMD code our synthesis produces and the robustness of the performance improvements on the various benchmark inputs.

Performance profiling of the 483.Xalan program shows that approximately 14% of the total runtime is spent executing the `find` algorithm on the cache. As our loop micro-benchmarks indicate that the SIMD `find` code is between 1.08× and 2.4× faster than the baseline implementation we would expect to see between a 1% (worst case) and 8% (best case) reduction in total runtime.

Table 2 shows the performance results obtained by replacing (by hand) the calls to the `find` algorithm with calls to our synthesized SIMD code. We show for each input provided in the SPEC test suite the size of the input and the percentage reduction in the total program execution time. The speedup indicates that the calls to the synthesized SIMD code are, depending on the inputs, 1.15

Input Data	Input Size	Improve(%)
test	28KB	5.5%
train	39MB	2%
ref	56MB	5%

Table 2. Runtime improvement(%) for 483.Xalan.

to 1.5 times faster than the standard implementations (matching our expectations from the micro-benchmark results). In contrast to the widely variable speedup (and slowdown) seen by changing the underlying data structure, the use of the vectorized find loop showed consistent improvements of 2%-5% across the inputs.

8. Related Work

Vectorization. Automatic program vectorization is a challenging problem which requires the application of a wide range of techniques for effective vectorization including: loop transformation [18, 29], control flow dependency elimination [18], alignment optimizations [28, 40], and finding sets of operations that can be executed in parallel [19, 33]. However, previous work on compiler auto-vectorization has focused on what are traditionally considered *regular* applications (e.g., scientific codes, multimedia applications, encode/decode algorithms) and on special purpose libraries (codecs, encryption, etc.), where loops have well behaved termination conditions, data sets are of a fairly regular/large size, and data layouts are suited to SIMD computation.

In contrast, the work in this paper seeks to apply SIMD instructions to *irregular* loops from standard library implementations which often have poor data layouts, small iteration count loops, and extensive data dependent control flow. These types of programs present different and in many ways more difficult problems to the automatic construction of vectorized code. These difficulties are highlighted in a recent study by Maleki et. al. [21] which examines a number of state of the art vectorizing compilers and their ability to vectorize a range of loops. They conclude that modern compilers fail to vectorize many loop patterns due to a lack of development resources needed to build a compiler that can identify and treat all the needed loop and computation patterns.

The work in this paper focuses on the issues of sub-optimal data layouts and complex data driven control flow but does not examine issues involving indirect memory accesses via pointers. Recent work has begun to explore how to reorganize and traverse pointer based structures into flat structures which are amenable to SIMD computation [31]. In particular work on unique pointer referencing [3, 20] and object lifetime, either as a global invariant or in a localized section of code, based on static [20, 22] is a critical first step dealing with the challenges posed by pointers.

Verification. Translation validation is a general method for checking a posteriori that compiler runs are correct, i.e output target programs that are semantically equivalent to input programs [30, 43]. Product programs reduce relational verification to functional verification of a single program: instances include self-composition [6], cross-products [42], and their combination [4, 5]. These methods are able to validate a wide range of loop optimizations, including those needed by our method. In this work, we use product programs to generate synthesis conditions for loop bodies.

Relational Hoare Logic is a generalization of Hoare logic in which judgments involve two programs, and pre- and post- conditions are denoting relations on states [8]. Relational Hoare Logic is effective for proving the correctness of structure-preserving optimizations, and simple optimizations that alter the control flow of programs. However, the core logic of [8] does not support the kind of loop optimizations required for our examples.

Synthesis The area of program synthesis is gaining renewed interest [10, 11, 35]. Srivastava et.al. introduced the notion of *proof-theoretic synthesis* where the problem of synthesizing a loopy program, given a pre/post condition, is reduced to the problem of simultaneously synthesizing loop-free fragments and loop invariants [38]. This approach is limited to synthesis of simple programs whose total correctness proofs or loop invariants can be expressed as simple templates. In contrast, we reduce the problem of vectorizing a given loopy program, to the problem of synthesizing only a loop-free fragment (without the need to synthesize any sophisticated loop invariants). This reduction is enabled by our use of the powerful relational verification methodology, which allows us to separate the process of verification and synthesis by generating an over approximation of the equalities required for equivalence proof.

The problem of synthesizing loop-free programs has been addressed in a variety of domains including bit-vector algorithms [12, 15, 36], ruler/compass based geometry constructions [13], text transformations [24], and algebraic proof problems [34]. One class of technique is based on *constraint solving*, which involves reducing the synthesis problem to that of solving a SAT/SMT formula (inside a CEGIS loop) and let an off-the-shelf SAT/SMT solver efficiently explore the search space. The applicability of this technique has been limited to semi-automatic settings, where the user provides templates [36] or reasonable over-approximation of the number of times each base component is used in the desired program [12]. Another class of technique is based on *brute-force search*, which involves systematically exploring the entire state space of artifacts and checking the correctness of each candidate. This approach often requires use of non-trivial optimizations and performs best when the specification consists of examples as opposed to a formal relational specification. Past work has included optimizations such as goal-directed search [13], clues based on textual features of examples [24], and common subexpression evaluation [34]. In this work, we combine the CEGIS loop from constraint-solving approaches with brute force search approach and novel optimizations.

Superoptimization is the task of finding an optimal code sequence for a straight-line target sequence of instructions, and it is used in optimizing performance-critical inner loops. One approach to superoptimization has been to constrain the search space to a set of equality-preserving transformations [2, 16], and then select the one with the lowest cost. This approach is limited by the kind of transformations that it can generate. Another approach to superoptimization has been to use brute-force search and enumerate sequences of increasing length or cost, testing each for equality with the target specification [23]. We also use brute-force search, but combined with a CEGIS loop and non-trivial optimizations.

9. Conclusion

This work presents a new approach to addressing the challenges that are present when attempting to harness the performance and power advantages available from data-parallel SIMD operations. In particular we looked at the problem of auto-vectorizing loops that have sub-optimal data layouts and complex data driven control flow, as is frequently the case in general purpose library code from the C++ STL or the C# Base Class Libraries. Our approach is driven by three core objectives: to produce an auto-vectorizer that is *applicable* to a wide range of *irregular* loops, that produces code which *reliably improves* the performance of the loop, and that guarantees the *correctness* of the resulting SIMD code.

These objectives led us to a novel auto-vectorization approach based on *deductive* loop rewriting and *inductive* synthesis of loop-free code. The use of inductive synthesis for constructing the loop body makes it particularly robust when dealing with the multitude of variations on the basic loop forms (find, map, reduce, etc.) that

appear in practice. In addition this approach allows us to produce correctness proofs for the resulting code. We believe that this underlying approach of combining deductive code restructuring with inductive code generation represents a general and promising way forward in research on program compilation. Thus, this work is an important step in both expanding the set of programs that can be automatically SIMDized and in the larger problem of effective compilation for specialized hardware.

Acknowledgments

We would like to thank the PPoPP reviewers and Rastislav Bodik for their constructive comments and thoughts on this work. This work was supported in part by: European Projects FP7-318337 ENTRA, FP7-231620 HATS and FP7-256980 NESSoS, Spanish project TIN2009-14599 DESAFIOS 10, Madrid Regional project S2009TIC-1465 PROMETIDOS. César Kunz is funded by Spanish Juan de la Cierva programme (JCI-2010-08550). Juan Manuel Crespo is funded by FPI Spanish programme (BES-2010-031271).

References

[1] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *OOPSLA*, 2000.

[2] S. Bansal and A. Aiken. Automatic generation of peephole superoptimizers. In *ASPLOS*, 2006.

[3] E. Barr, C. Bird, and M. Marron. Collecting a Heap of Shapes. Technical Report MSR-TR-2011-135, Microsoft Research, Dec. 2011.

[4] G. Barthe, J. M. Crespo, and C. Kunz. Relational verification using product programs. In *FM*, 2011.

[5] G. Barthe, J. M. Crespo, and C. Kunz. Beyond 2-safety: Asymmetric product programs for relational program verification. In *LFCS*, 2013.

[6] G. Barthe, P. R. DArgenio, and T. Rezk. Secure information flow by self-composition. In *CSFW*, 2004.

[7] M. Bebenita, F. Brandner, M. Fahndrich, F. Logozzo, W. Schulte, N. Tillmann, and H. Venter. SPUR: A trace-based JIT compiler for CIL. In *OOPSLA*, 2010.

[8] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *POPL*, 2004.

[9] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *PLDI*, 2005.

[10] S. Gulwani. Dimensions in program synthesis. In *PPDP*, 2010. Invited talk paper.

[11] S. Gulwani. Synthesis from examples: Interaction models and algorithms. *SYNASC*, 2012. Invited talk paper.

[12] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *PLDI*, 2011.

[13] S. Gulwani, V. A. Korthikanti, and A. Tiwari. Synthesizing geometry constructions. In *PLDI*, 2011.

[14] Intel Optimization Manual (June 2011) – Section 6.5.1. http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf.

[15] S. Jha, S. Gulwani, S. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *ICSE*, 2010.

[16] R. Joshi, G. Nelson, and K. H. Randall. Denali: A goal-directed superoptimizer. In *PLDI*, 2002.

[17] C. Jung, S. Rus, B. P. Railing, N. Clark, and S. Pande. Brainy: Effective selection of data structures. In *PLDI*, 2011.

[18] K. Kennedy and J. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., 2002.

[19] S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *PLDI*, 2000.

[20] K.-K. Ma and J. Foster. Inferring aliasing and encapsulation properties for java. In *OOPSLA*, 2007.

[21] S. Maleki, Y. Gao, M. Garzarán, T. Wong, and D. Padua. An evaluation of vectorizing compilers. In *PACT*, 2011.

[22] M. Marron. Structural analysis: Shape information via points-to computation. Technical Report 1201.1277, arXiv, Jan. 2012.

[23] H. Massalin. Superoptimizer - a look at the smallest program. In *ASPLOS*, 1987.

[24] A. Menon, O. Tamuz, S. Gulwani, B. Lampson, and A. Kalai. A machine learning framework for programming by example. In *ICML*, 2013.

[25] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing wrong data without doing anything obviously wrong! In *ASPLOS*, 2009.

[26] G. Necula. Proof-carrying code. In *POPL*, 1997.

[27] G. Necula and P. Lee. Safe kernel extensions without run-time checking. In *OSDI*, 1996.

[28] D. Nuzman, I. Rosen, and A. Zaks. Auto-vectorization of interleaved data for SIMD. In *PLDI*, 2006.

[29] D. Nuzman and A. Zaks. Outer-loop vectorization: Revisited for short SIMD architectures. In *PACT*, 2008.

[30] A. Pnueli, M. Siegel, and F. Singerman. Translation validation. In *TACAS*, 1998.

[31] B. Ren, G. Agrawal, J. Larus, T. Mytkowicz, T. Poutanen, and W. Schulte. SIMD parallelization of applications that traverse irregular data structures. In *CGO*, 2013.

[32] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *ESEC/FSE-13*, 2005.

[33] J. Shin, M. Hall, and J. Cha. Superword-level parallelism in the presence of control flow. In *CGO*, 2005.

[34] R. Singh, S. Gulwani, and S. Rajamani. Automatically generating algebra problems. In *AAAI*, 2012.

[35] A. Solar Lezama. *Program Synthesis By Sketching*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2008.

[36] A. Solar-Lezama, R. M. Rabbah, R. Bodík, and K. Ebcioglu. Programming by sketching for bit-streaming programs. In *PLDI*, 2005.

[37] SPEC. Standard Performance Evaluation Corporation (SPEC). http://www.spec.org/cpu2006/.

[38] S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. In *POPL*, 2010.

[39] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem: Overview of methods and survey of tools. *ACM TECS*, 7(3), 2008.

[40] P. Wu, A. Eichenberger, and A. Wang. Efficient SIMD code generation for runtime alignment and length conversion. In *CGO*, 2005.

[41] K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, and P. Stodghill. Is search really necessary to generate high-performance BLAS? *Proceedings of the IEEE*, 93(2), 2005.

[42] A. Zaks and A. Pnueli. Covac: Compiler validation by program analysis of the cross-product. 2008.

[43] L. D. Zuck, A. Pnueli, and B. Goldberg. Voc: A methodology for the translation validation of optimizing compilers. *J. UCS*, 9(3), 2003.