Probabilistic Source-Level Optimisation of Embedded Programs

Björn Franke, Michael O'Boyle, John Thomson, and Grigori Fursin
University of Edinburgh
Institute for Computing Systems Architecture (ICSA)
Edinburgh, EH9 3JZ, United Kingdom
{bfranke,mob}@inf.ed.ac.uk,{John.Thomson,g.fursin}@ed.ac.uk

ABSTRACT
Efficient implementation of DSP applications is critical for many embedded systems. Optimising C compilers for embedded processors largely focus on code generation and instruction scheduling which, with their growing maturity, are providing diminishing returns. This paper empirically evaluates another approach, namely source-level transformations and the probabilistic feedback-driven search for "good" transformation sequences within a large optimisation space. This novel approach combines two selection methods: one based on exploring the optimisation space, the other focused on localised search of good areas. This technique was applied to the UTDSP benchmark suite on two digital signal and multimedia processors (Analog Devices TigerSHARC TS-101, Philips TriMedia TM-1100) and an embedded processor derived from a popular general-purpose processor architecture (Intel Celeron 400). On average, our approach gave a factor of 1.71 times improvement across all platforms equivalent to an average 41% reduction in execution time, outperforming existing approaches. In certain cases a speedup of up to ≈ 7 was found for individual benchmarks.

1. INTRODUCTION
High performance and short time to market are two of the major factors in embedded systems design. We want the end product to deliver the best performance for a given cost and we want this solution delivered as quickly as possible. In the past digital signal processing and media processing relied on hand-coded assembler programming of specialised processors to deliver this performance. However, as the cost of developing an embedded system becomes dominated by algorithm and software development, the use of high-level programming as a means of reducing time to market is now commonplace.

High-level programming in languages such as C, however, can lead to less efficient implementations when compared to hand-coded approaches [23]. Therefore, there has been a large amount of research interest in improving the performance of optimising compilers for embedded systems, e.g. [16]. Such work largely focuses on improving back-end, architecture specific compiler phases such as code generation, register allocation and scheduling. However, the investment in ever more sophisticated back-end algorithms produces diminishing returns [8].

Given that an embedded system typically runs just one program for its lifetime, we can afford much longer compilation times (e.g. in the order of several hours) than in general-purpose computing. In particular, feedback directed or iterative approaches where multiple compiler optimisations are tried and the best selected has been an area of recent interest [4, 18]. However, these techniques still give relatively small improvements as they effectively restrict themselves to trying different back-end optimisations.

In this paper we consider an entirely distinct approach, namely using source-level transformations for embedded systems. Such an approach is by definition highly portable from one processor to another and is entirely complementary to the efforts of the manufacturers back-end optimisations. In fact, we show that it allows vendors to put less effort into their compiler reducing the time to market of their product, while giving higher performance (see section 5.3.3).

While high-level approaches can deliver good performance, it is extremely difficult to predict what the best transformation should be. It depends both on the underlying processor architecture and the native compiler. Small changes in the program, a new release of the native compiler or the next generation processor will all impact on the transformation selection. Typically, high level restructures have a static simplified model [3] with which to guide transformation selection. It has been shown [5, 9], however, that the optimisation space is highly non-linear and that such completely static approaches are doomed to failure.

In this paper we propose a new approach to high-level transformation – namely probabilistic optimisation. Essentially, we use stochastic methods to select the high-level transformations directed by execution time feedback where we trade off optimisation space coverage against searching in known

LCTES '05 Chicago, Illinois, USA

(a) Original implementation

```
void lmsfir(float input[], float output[],
float expected[], float coefficient[], float gain)
{
  int i;
  float sum,term1,error,adapted,old_adapted;

  sum = 0.0;
  for (i = 0; i < NTAPS; ++i) {
    sum += input[i] * coefficient[i];
  }
  output[0] = sum;
  error = (expected[0] - sum) * gain;
  for (i = 0; i < NTAPS-1; ++i) {
    coefficient[i] += input[i] * error;
  }
  coefficient[NTAPS-1] = coefficient[NTAPS-2] +
    input[NTAPS-1] * error;
}
```

(b) TS-101 implementation

← Loop totally unrolled
← Array references dismantled

← Loop totally unrolled
← Array references dismantled

(c) TriMedia implementation

← New temps. introduced

← Lowered to DO-WHILE loop*
← Pseudo 3-address code
← Linear pointer-based array traversal

← Loop totally unrolled
← Pseudo 3-address code
← Linear pointer-based array traversal

* See figure 2 for the specific code of this loop.

Figure 1: Differences between the original *lmsfir* implementation (a), and implementations for the TigerSHARC (b) and TriMedia (c) processors

good regions. Using such an approach we achieve remarkable performance improvements - on average a 1.71 speedup across three machines. We demonstrate that our approach can automatically port to any new processor and extract high levels of performance, unachievable by traditional techniques, with no additional native compiler effort.

The paper is organised as follows. Section 2 provides a motivating example demonstrating the need for searching high level transformations. Section 3 describes the transformation space considered and is followed in section 4 by an overview of the search techniques used. This is followed in section 5 by an empirical evaluation of our approach. Section 6 surveys related work and is followed by some concluding remarks in section 7.

2. MOTIVATION & EXAMPLE

High-level transformations are a portable, yet highly effective way to improve performance by enabling the native compiler to produce efficient code. Deriving efficient program transformation sequences, however, is a complex task. For all but the most basic programs, the interaction between the source-to-source transformation, the native compiler and its built-in optimisations and the underlying target architecture cannot be easily analysed and exploited [3]. Furthermore, programmers frequently apply a series of program transformations to the program based on their expert knowledge and experience with a specific processor and its compiler. However, with each new generation of the processor or even release of a new compiler version their knowledge becomes outdated. Furthermore, new processors and their frequently immature compilers are a challenge for any program developer aiming at high performance.

As an example, consider the program excerpt in figure 1(a). The *lmsfir* function is part of the UTDSP [15] *LMSFIR* benchmark. It computes a single point of an N-tap adaptive finite impulse response (FIR) filter applied to a set of input samples. The first of the two *for* loops iterates over the input and coefficient vectors and performs repeated multiply-accumulate (MAC) operations. The second loop updates the filter coefficient for the next run of this filter function.

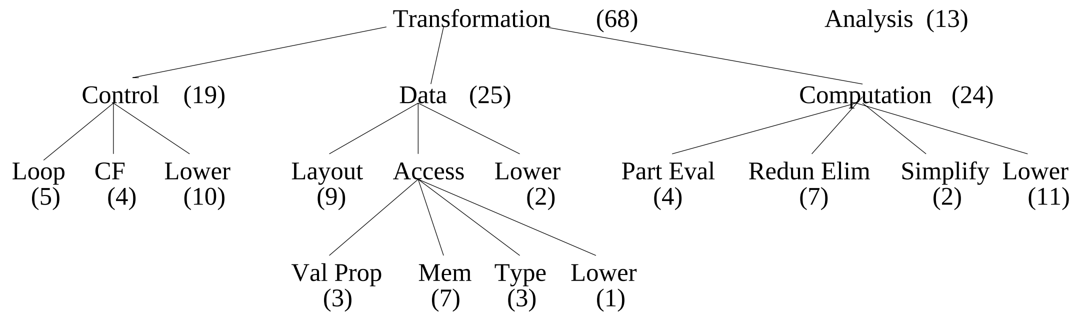
```
data = input; coef = coefficient; sum = 0.0F;
i = 0;
do
  {
    {
      float *suif_tmp, *suif_tmp0;
      suif_tmp = data;
      data = data + 1;
      term1 = *suif_tmp;
      suif_tmp0 = coef;
      coef = coef + 1;
      term2 = *suif_tmp0;
      sum = sum + term1 * term2;
    }
    i = i + 1;
  } while (!(8 <= i));
```

Figure 2: First loop of example 1(a) optimised for the TriMedia processor

In figure 1(b) the main differences due to transformations in an optimised TigerSHARC implementation are listed. While the routine has not changed semantically, it outperforms the routine in figure 1(a) by a factor of 1.75 on the TigerSHARC TS-101 processor. In this transformed version of the program, both loops have been flattened and the array references been dismantled into explicit base address plus offset computations.

On the TriMedia, however, different transformations produce the best performing *lmsfir* implementation (see figure 1(c)). Here the speedup of 1.2 is achieved by converting the first *for* loop into a *do-while* loop and flattening the second. All array references have been converted to pointers and an almost 3-address code produces the best result. The first loop of example 1(a) in its optimised form for the TriMedia is shown in figure 2.

This short example demonstrates how difficult it is to predict the best high-level transformation for a new platform. Feedback-directed compilers interleave transformation and



Numbers in parentheses denote the number of transformations in corresponding transformation category.

Figure 3: Transformation categorisation

profiled execution stages to actively search for good transformation sequences. Portable, optimising compilers, however, must be able to search a potentially huge transformation space in order to find a successful sequence of transformations for a particular program and a particular architecture. In this paper we propose a probabilistic search algorithm that is able to examine a small fraction of the optimisation space and still find significant performance improvements.

3. OPTIMISATION SPACE

Transforming or rewriting a program at source level has an impact on performance. To illustrate this, consider each of the UTDSP benchmarks described in figure 5 which are supplied in up to four distinct versions. Firstly each is available in either an array or pointer form. In addition, each of these may also be available in source-level software pipelined forms. Although these versions are four independent sources, each version can readily be derived from the other by pointer conversion/recovery [17, 8] or source-level software-pipelining [20]. Figure 4 shows the average execution time of each version across the benchmarks on each processor. On the TigerSHARC the clean array version gives the best average performance while the TriMedia prefers the pointer based version. In this paper, in order to compare the effect of native compiler on system performance, we consider two compilers, GCC and ICC, for the Celeron. Both compilers marginally prefer the array based code over pointer based versions. In most cases, with the notable exception of the TriMedia, the software pipelined versions of the program perform poorly. From this set of data, we can conclude that source-level transformations will affect performance and that this will depend on the processor, program and possibly the underlying compiler.

Due to the variation in performance of the four different versions, *all speedups in this paper are with respect to the best performing original code.* For example, in the case of the TigerSHARC this is normally the array based original code while on the TriMedia it is usually the pointer version.

Selecting the best overall high-level transformation normally consists of selecting a sequence of smaller transformations which are applied to part or all of the program. Given that certain transformations may be parameterised[1], and that

[1]For example, loop unrolling is parameterised by the unroll factor.

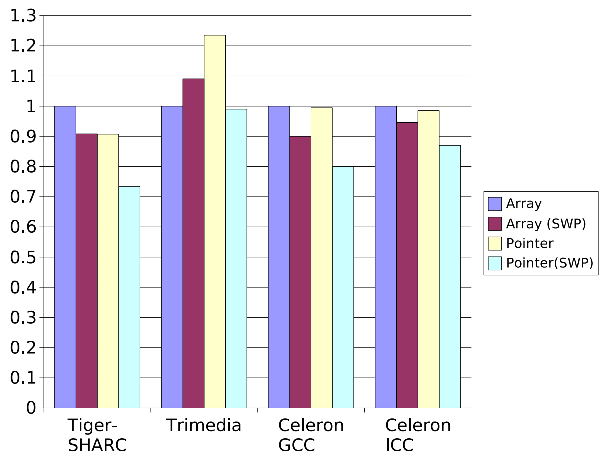


Figure 4: Relative speedup or slowdown for different coding styles per processor. The data is normalised to the performance of the baseline array code

different combinations may be considered, selecting the best transformation is effectively an optimisation problem over the space of all possible transformations.

In the remainder of this section, we describe the source level transformation considered and novel mechanism to encode all transformations as binary decisions, allowing existing optimisation techniques to be used.

3.1 Transformations

In this paper we consider 81 high level transformations, applicable to C programs and available within an extended SUIF [11] based framework. For convenience we have classified them as shown in figure 3. 13 are in effect analysis phases that mark the IR enabling later transformations which actually modify the source. These transformations can be classified into three broad groups; those aimed at modifying the program's control-flow, those that modify the actual computation performed and those focused on data which is further subdivided into actual layout and access. These broad categories are further refined as shown in figure 3.

All categories contain lowering transformations which translate a complex structure into a smaller one, i.e. unpacking a structure into its sub-components.

The control-flow transformations are aimed either at loop transformations or more general control-flow changes. The data access transformations include value propagation, modifying memory references and data type conversion. Finally, the computation based transformations include partial evaluation, redundancy elimination and code simplification. This is by no means a definitive transformation taxonomy, but provides an overview of the options available.

3.2 Encoding Transformations

One of the main difficulties in selecting the best transformation sequence is that many transformations are position dependent, i.e. only applied to a part of the program. Unlike global optimisations, we have to specify the location of the transformation. Furthermore, these transformations may be parameterised. This leads to two problems. Firstly, the optimisation space now increases in size and, secondly, it becomes asymmetric in description. This means that we cannot search the space in a uniform manner.

To overcome this we have devised a simple method to make the treatment of parameterised location specific transformations indistinguishable from the yes/no decision of global optimisations such as constant propagation. This is achieved by simply enumerating all possible parameters and all locations. For example consider the case of loop unrolling. If there are three loops each of which may be unrolled up to eight times then there are 24 possible loop unroll transformations to consider. At the beginning of each optimisation we determine the total number of transformations needed based on the program size and the parameter space selected. As the number of loops may change over time due to, say, fusion or distribution, then the number of transformations to select from changes dynamically. In the case of the focused search described below, we therefore need to track an individual transformation's contribution to success where the number of transformations changes over time.

4. SEARCH

For all but very small sets of transformations it is impossible to perform an exhaustive search of all possible transformation sequences up to a given short length and of all parameters. Any practical search algorithm must therefore find a "good" rather than an optimal solution in acceptable time.

In figure 6 an overview of an iterative compilation and optimisation framework is given. C code enters the system and is translated into an intermediate representation on which all transformations operate. After finishing the transformation process, the IR is translated back into C code and compiled into an executable by the particular machine's native C compiler making use of its most aggressive optimisation setting.

We have implemented this transformation toolkit and use the Stanford SUIF compiler [11] to provide us with a C front-end, a code generator and a rich number of already implemented transformations.

Program	Description
fft_1024 **fft_256**	Radix-2, in-place, decimation-in-time Fast Fourier Transform (FFT)
fir_256_64 **fir_32_1**	Finite Impulse Response (FIR) filter
iir_4_64 **iir_1_1**	Infinite Impulse Response (IIR) filter
latnrm_32_64 **latnrm_8_1**	Normalised lattice filter
lmsfir_32_64 **lmsfir_8_1**	Least-mean-squared (LMS) adaptive FIR filter
mult_10_10 **mult_4_4**	Matrix multiplication
G721_encoder	ITU ADPCM speech transcoder
G721_decoder	ITU ADPCM speech decoder
V32.modem encod.	V.32 modem encoder
V32.modem decod.	V.32 modem encoder
compress	Image compression using Discrete Cosine Transform
edge_detect	Edge detection using 2D convolution and Sobel operators
histogram	Image enhancement using histogram equalisation

Figure 5: UTDSP benchmarks

4.1 Optimisation Algorithm

Central to the iterative transformation framework is an optimisation algorithm hosted by the optimisation engine in figure 6. The huge size of the optimisation space and its complexity make it necessary to find a balanced trade-off between *space exploration* and *focused search*. For the benchmarks considered here the size of the space is approximately 10^{90}. To find good points, whilst keeping the number of sample points (and thus the number of program runs) within reasonable limits, we employ probabilistic algorithms.

Although the random based search of the optimisation space leads to significant performance improvement [9], it is by definition unable to direct efforts and search for an optimal point. If a transformation or sub-sequence is found to consistently perform well or poorly or indeed have no effect, we would like to use this information to guide the search. However, there is a natural tension between avoiding hardwiring of biased heuristics and cost-effective search. What is needed is a technique that combines an unbiased sampling of the transformation space and with feedback focused attention on good areas.

In order to overcome this dilemma of space exploration vs. focused search, we have combined two simple, yet powerful algorithms representing each of the two domains. These two algorithms compete with each other and within a final merge stage the best of the two individual solutions is chosen. To facilitate a broad and non-biased space coverage we have chosen a simple *random search* as our space exploration algorithm. The focused search is represented by a *machine learning* algorithm based on a modified *Population-based Incremental Learning (PBIL)* [1] approach. Both algorithms can be considered as two extreme cases of a continuum where the learning rate is $LR = 1$ for the PBIL inspired technique

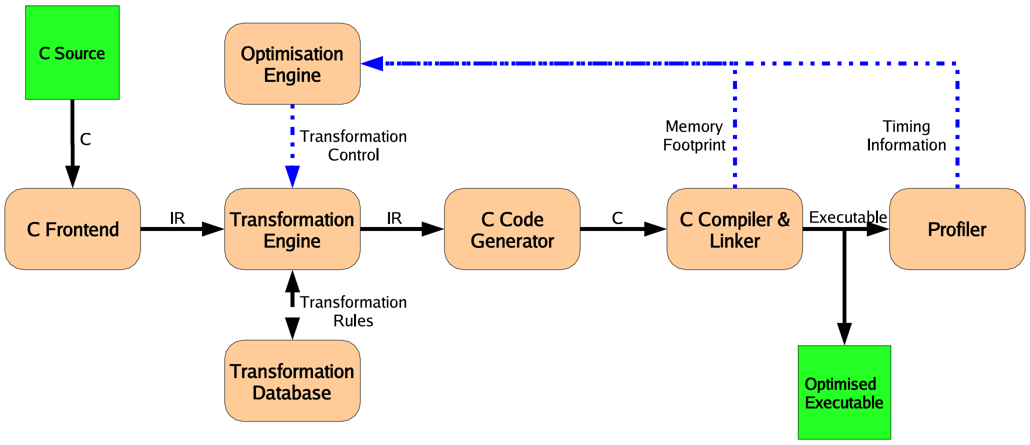


Figure 6: Overview of the iterative compilation/optimisation framework

and $LR = 0$ for random search. In particular, in a competitive learning network the activation of the output units are computed and the weights adjusted according to the rules given by the following two equations [1]:

$$output_i = \sum_j w_{ij} \times input_j \tag{1}$$

$$\Delta w_{ij} = LR \times (input_j - w_{ij}) \tag{2}$$

A learning rate $LR = 0$ leads to constant weights which are not adjusted during the search. On the other hand, a learning rate $LR = 1$ enforces strong adjustment to the individual weights over changing input. In the two following two paragraphs we discuss both algorithms in detail.

4.2 Space Exploration

Random search assigns a constant uniform probability distribution to the set of transformations and choses the next transformation solely based on a value generated by a pseudo-random number generator. In the case of parameterised transformations, we equally divide the assigned probability across all enumerated versions. For example if each transformation has a 0.1 probability of being selected but there are 50 loop unrolling options, then each of them is assigned a probability of 0.002.

Formally, the learning rate is $LR = 0$ for random search as no information is carried across iterations of the algorithm and from equation 2 it follows that $\Delta w_{ij} = 0$.

Both the transformation and the length of the transformation sequence (up to some upper limit) are determined by a random process. The random search algorithm does not use the effectiveness of any transformations to direct its search.

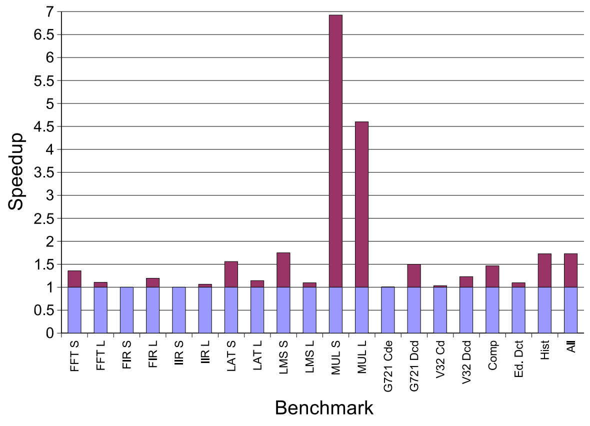


Figure 7: Speedup due to high-level transformation over the most aggressive native compiler optimisation for the TigerSHARC.

4.3 Focused Search

PBIL is a stochastic search technique that aims at integrating genetic algorithms and competitive learning. It increases the probability of an option being selected whenever a positive instance using that option is encountered.

In our stochastic optimisation algorithm, transformations have an associated selection probability, but unlike the space exploring random search algorithm, probabilities can change over time and their distribution does not need to be uniform, i.e. $LR \neq 0$. In fact, we have chosen $LR = 1$ to emphasise its fast convergence on encountered performance enhancing transformations. The original PBIL algorithm considers binary encodings of parameters and generates a population of solutions based on a fixed-length probability vector, which had to be modified for our purposes.

Starting with a uniform probability distribution, individual sample points (i.e. transformation sequences) are chosen and evaluated by executing the corresponding program. The selection probabilities of the individual transformations are updated based on the success (i.e. execution time) of the sequence as a whole. Transformations contributing to better performance are rewarded while those resulting in performance losses are penalised. Thus, future sample points are more likely to include previously successful transformations more frequently and search their neighbourhood more intensively.

Standard PBIL allows for random mutation within the probability vector, but we discard this as we do not wish to incur the overhead. Finally we do not generate a population based on a probability vector, but just one candidate. Depending on its success we update the probability vector accordingly.

The high learning rate, lack of mutation and a single candidate per generation means that we strongly focus the search based on feedback results.

5. EXPERIMENTAL EVALUATION

In this section we present and analyse the empirical results we gained with our tool. All results are found after running the search algorithm for 500 evaluations corresponding two about 2-6 hours search.

5.1 Processors and Compilers

We have evaluated our adaptive optimisation scheme against three different processors representing different aspects of the embedded computing domain. Among the three embedded processors are a high-performance floating-point digital signal processor (Analog Devices TigerSHARC TS-101), a multimedia processor (Philips TriMedia TM-1100) and an embedded processor derived from a popular general-purpose processor architecture (Intel Celeron 400).

As native compilers we used Analog Devices' VisualDSP++ 3.5 for the TigerSHARC v7.0.1.5, Philips' TriMedia v1.1y Software Development Environment (SDE v5.3.4) for the TriMedia, and both Intel's ICC 8.0 and the GNU GCC 3.3.3 for the Celeron. The highest optimisation settings were used on the native compilers and execution times were measured using hardware cycle counters.

The optimisation methodology and transformation toolkit are highly portable and have been ported within few hours to eight distinct embedded processor architectures. However, as the time of writing our experimental data for these additional platforms is not yet available.

5.2 Benchmarks

We have chosen the *UTDSP* [15, 19] benchmark suite to evaluate our technique. This set of benchmarks contains compute-intensive DSP kernels as well as applications composed of more complex algorithms and data structures. The details are shown in figure 5. Many of the programs are available in up to four coding styles (explicit vs pointer-based array references, plain vs source-level software pipelined). Some of the benchmarks are excluded from this study. This is due to the incompatibility between the differing interpretations of acceptable C syntax/semantic between SUIF and the native compilers. The TigerSHARC in particular is much stricter than SUIF in terms of the C accepted. Also some of the benchmarks are focused on bit manipulation which causes problems due to conflicting endianness. This issue has been fixed recently, however, data is not yet available.

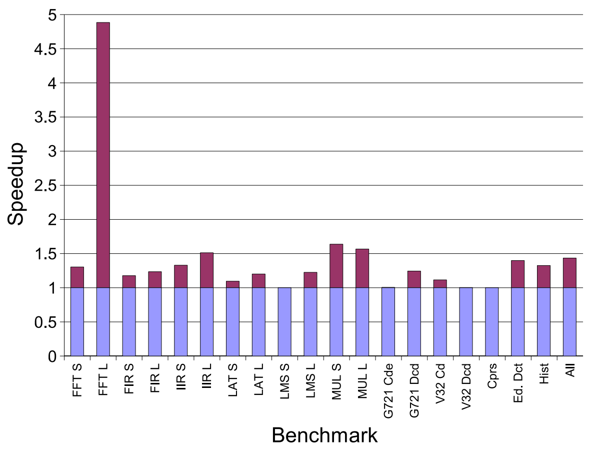


Figure 8: Speedup due to high-level transformation over the most aggressive native compiler optimisation for the TriMedia.

5.3 Results

As stated in section 4, all speedups are with respect to the best performing original program giving a true evaluation of our approach. Thus, the best original execution time of the four possible versions of each program was selected for speedup comparison using the highest optimisation level selected on the native compiler.

5.3.1 *Platform Based Evaluation*

Figures 7, 9, 10 and 8 show the performance improvements achieved by our approach across processors and benchmarks. All the platforms benefited from the iterative search. The TigerSHARC had an average speedup of 1.73, the TriMedia 1.43, the Celeron with GCC 1.54 and with ICC 2.14 giving an overall average of 1.71. This overall figure demonstrates the importance of high-level optimisation. In other words, using a platform independent approach we are able to reduce execution time on average by 41%, outperforming any other approach.

Examining the TigerSHARC results (see figure 7) more closely we see there is much variation. Surprisingly, the matrix multiplication routines can be improved by almost a factor of 7 by completely flattening the code. As this is such a well known routine, one would have thought that the baseline compiler would do well here.

The iterative scheme performs less well on the very small data sizes of *FIR* and *IIR*, unlike the other processors. It also is unable to improve the performance of the *G721 encoder*, a problem shared by all of the processors.

A different picture emerges when considering the Celeron

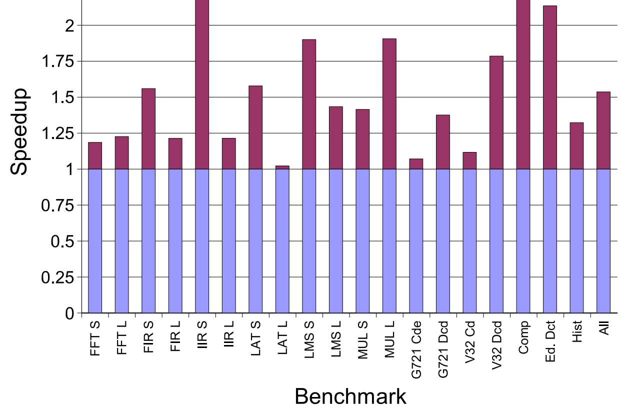


Figure 9: Speedup due to high-level transformation over most aggressive native compiler optimisation for the Celeron/GCC.

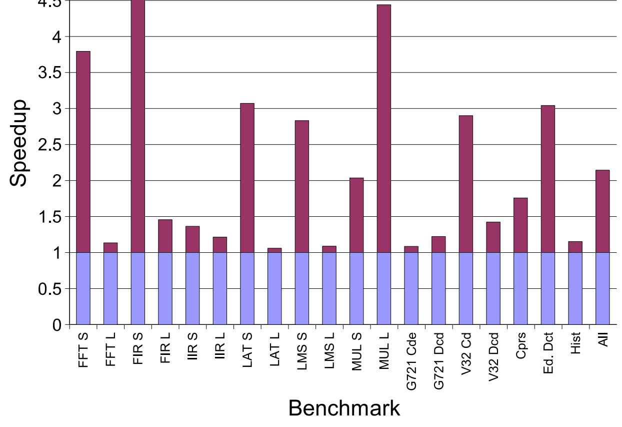


Figure 10: Speedup due to high-level transformation over most aggressive native compiler optimisation for the Celeron/ICC.

processor with GCC (see figure 9) where the speedups are less variable. In direct contrast to the TigerSHARC, large performance gains are achieved on the small data sized *IIR* program. Good results are also found for the *compression* and *edge detection* applications. Like the TigerSHARC, little performance was gained on the *G721 encoder*.

The largest performance gains were achieved with the ICC compiler on the Celeron. This in itself is a surprising result given that it is the most mature compiler here and therefore should have proved difficult to improve upon. Like the TigerSHARC it performs well on the large matrix multiplication and the small *FFT* and poorly on the *G721 encoder*. However, it performs well on the small *IIR* like GCC and shares similar performance gains on *edge detection* and *V.32 encoder*. We will compare the two compilers GCC and ICC for the Celeron in more detail below (see section 5.3.3).

The TriMedia has the lowest average speedup of 1.43 and like the TigerSHARC has an uneven distribution of results with the large *FFT* achieving a speedup of almost 5. Once again it performs poorly on the *G721 encoder*, but unlike other platforms it performs poorly on the *V.32 decoder* and *compress* benchmarks.

5.3.2 Benchmark Orientated Evaluation

If we examine the average performance improvement across the benchmarks as shown in figure 11, we see that only three of them fail to achieve an average speedup of 1.25. *LATNRM* benefits from loop unrolling. However, due to cross-iteration dependences the native compilers instruction scheduler cannot take full advantage of the enlarged loop body. *LMSFIR* suffers from a coding style that introduces frequent conditional branches to the innermost loop. Similarly, *G721* is limited in its transformation potential by many conditional branches between tiny basic blocks.

Surprisingly in four out of six cases high-level iterative search is able to speed up programs to a greater extent for small rather than large data sizes. This is counterintuitive as many of the restructuring transformations only have any noticeable effect when dealing with large amounts of data and

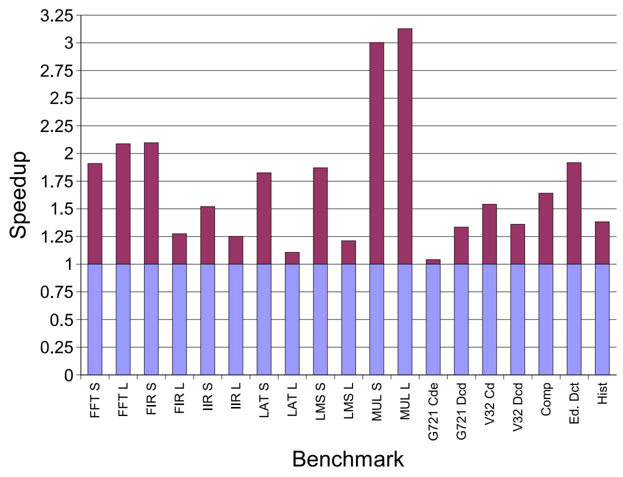


Figure 11: Program speedup averaged across all platforms over the most aggressive native compiler optimisation.

computation. Examining the output code, it seems that in several cases the iterative search has completely unrolled or flattened certain sections of code turning loops into large basic blocks and act as an enabler of baseline compiler optimisation. This is the reason for the large speedup of matrix multiplication on the TigerSHARC.

5.3.3 GCC vs ICC

Using two compilers on one platform gives an insight in to their effect on performance. As expected, overall the ICC compiler outperforms GCC and is approximately 1.22 times faster on average. However, on applying high-level transformations on top of GCC, we see an improvement on average of 1.54, outperforming ICC on its own. This means that an automatic platform-independent approach can use simple compilers as a baseline and outperform hand-crafted optimisers based on many person years work. Furthermore, it allows vendors to put less effort into their compiler reducing the time to market of their product, while giving higher

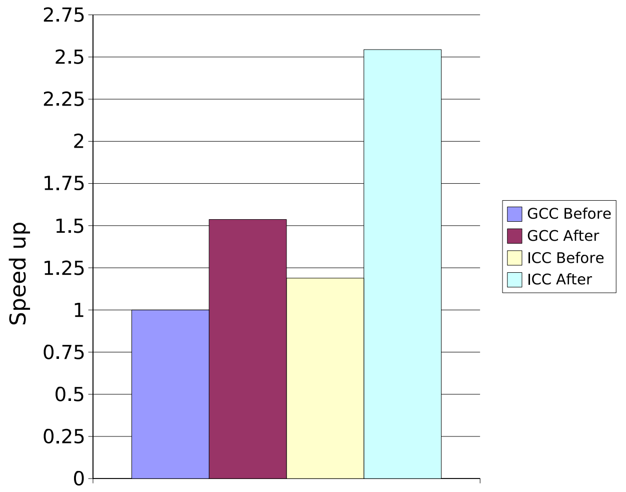
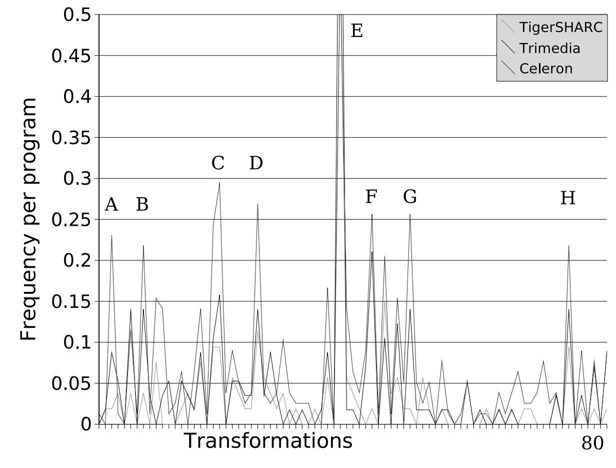


Figure 12: Comparison of the two compilers for the Celeron. Results are normalised to GCC performance before transformation.



A - Break up large expression trees, B - Value propagation, C - Hoisting of loop invariants, D - Loop normalisation, E - Loop unrolling, F - Mark constant variables, G - Dismantle array instructions, H - Eliminating copies.

X-axis: Enumerated transformations

Y-axis: Likelihood of transformation being selected

Figure 13: Probability of transformation being successful across all programs and processors.

performance.

The diagram also shows that applying transformations to ICC gives a speedup of more than 2.5 relative to GCC alone. This also shows that a platform-independent approach can also port and scale with improved baseline improvements and is a complementary approach to vendor improvements.

5.4 Transformations

Overall, loop transformations have been identified as the most beneficial class of transformations in our framework. This category (cf. figure 3) is followed by value propagation transformations and partial evaluation. The differences between the remaining classes are too small to derive any significant metrics from them.

Across all platforms and benchmarks, the focused search phase of the optimisation algorithm finds the best sequence 65% of the time with an average effective transformation sequence length of 4.1. For example, in *compress* on the TriMedia the best sequence of transformations was hoist loop invariants, optimise function parameter passing, globalise constants, scalarisation and flatten the main loop.

The remaining 35% of the time the best transformation sequence was found by the random phase where the absolute length was on average 40.1. This result looks surprising at first. No high-level restructuring compiler research has suggested that such sequence lengths are beneficial and they obviously contrast with the focused search results. However, as we are randomly selecting sequences between 1 and 80, then an average around 40 is to be expected. Furthermore, on examination it can be seen that there are many transformations included which do have any impact on the code. These junk transformation sub-sequences frequently contain repeated transformations or ones which have no effect on that particular program. Hence, the effective transformations sequence length is much shorter. This means that while long sequences may be beneficial, it is sufficient for future work to consider short but effective sequences, less than ten in length.

It is interesting to note that while the focused search finds the best optimisation 65% of the time, it achieves an average performance gain of 1.57. Random space exploration finds the best solution less often, but achieves an average speedup of 2.00 in these cases, justifying the choice of using two approaches to searching the space.

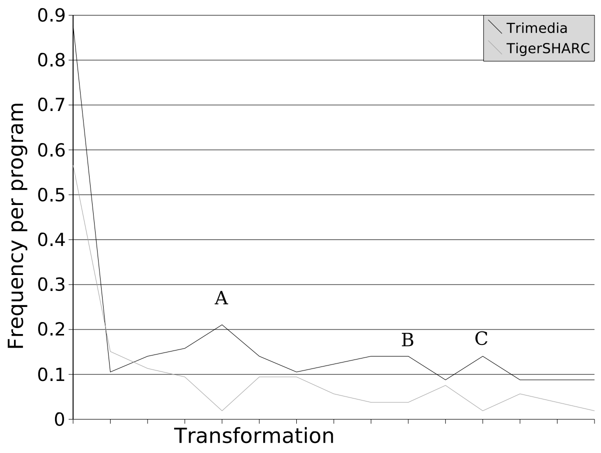
5.4.1 *Transformation Sequences*

Our experimental data confirms that it is hard to establish generally successful sequence orderings as they vary drastically. As a general rule, large increases in performance appear to come from one of two things – either loop unrolling or a large combination of other transformations (> 5). However, there appears to be little correlation between the length of the transformation sequence and the performance achieved.

5.5 Distribution

If we examine the probability distribution of the useful transformations across all processors and programs, there are eight transformations or peaks labelled A-H in figure 13. At first glance there seems to be much commonality across the processors. Loop unrolling (E) is by far the most successful transformation. Now, although it is well known to improve performance, it is surprising that it is so successful here as each of the native compilers applies unrolling internally[2]. This means that the heuristic employed by the native compiler is in fact poor. Propagating known values (B) and loop hoisting (C) are also useful transformations again surprising as a native compiler should perform this. Less obviously, breaking up expression trees (A) so that they can

[2]As we use the highest optimisation level available for each native compiler.



A - Data layout transformation, B - Control flow simplification, C - Dismantle array references. Transformations reordered so that the most effective transformations are leftmost. Only the first 14 significant transformations are shown.

Figure 14: Highlighted differences in overall effectiveness of transformations.

be effectively handled by the code generator proved useful. Finally changing arrays into pointer traversal (G) is useful for machines with separate address generation units while eliminating copies (H) reduces memory bandwidth.

If we focus now just on the TriMedia and TigerSHARC whose speedup profiles are similar, then we see that there are also differences among the processors. Figure 14 shows the transformation ordered by overall effectiveness. At three points A, B and C we see marked differences in the usefulness of transformations. Data layout transformation (A) rearranges the order and location of data declarations enabling the user more efficient addressing modes. This transformation is important for the TriMedia as this processor/compiler pair seems to be very sensitive to memory layout changes. Control flow simplification (B) eliminates redundant conditional branches and loops that might have been introduced by previous passes. Unlike the TigerSHARC with its dynamic branch predictor, unnecessary branching is very expensive for the TriMedia. Array reference dismantling makes the address computation of an array reference explicit and its importance to the TriMedia can be attributed to its compiler's relative immaturity.

5.6 Efficiency

Although we evaluate each benchmark 500 times in 2-6 hours this is acceptable in an embedded context where the cost is amortised over multiple runs. In fact on average, the majority of the performance improvement occurs within less than 200 runs. Future work which exploits program structure to guide transformation selection should further improve on this. Other possibilities include the consideration other learning rates different from 0 and 1.

6. RELATED WORK

6.1 Source-level program transformation

One major difficulty in the use of high-level transformations is that the preferred application language for embedded systems is C, which is not very well suited to optimisations. Extensive usage of pointer arithmetic [17, 23] prevents the application of well developed array-based data-flow analyses and transformations. Previous work [8], however, has shown that many pointer-based memory references can be eliminated and converted to explicit array references amendable to advanced analyses and transformations.

There has been limited work in the evaluation of high-level transformations on embedded systems performance. In [2] the trade-off between code size and execution time of loop unrolling has been investigated and in [12] the impact of tiling on power consumption has been evaluated. The impact of several high-level transformations on the DSPstone [23] kernel benchmarks is empirically evaluated on four different embedded processors in [8].

6.2 Feedback-directed program transformation

Iterative or adaptive compilation is a more recent development and has led to a number of publications in the past few years. Early work in this field [2, 13] investigate the iterative search for good parameters to loop unrolling and tiling. In [9], a random search strategy for numerical Fortran algorithms is evaluated, and [7] proposes neural network based search and optimisation, however, without giving empirical results. A partially user-assisted approach to select optimisation sequences is VISTA [14]. It combines user guides and performance information with a genetic algorithm to select local and global optimisation sequences. ADAPT [22] is a compiler-supported high-level adaptive compilation system. While it is very flexible and can be re-targeted to new platforms, it requires the compiler writer to specify heuristics for applying optimisations dynamically at runtime. Code optimisation at runtime, however, is usually not suitable in an embedded systems context. Other authors [18, 10, 4] have explored ways to search program- or domain-specific command line parameters to enable and disable specific options of various optimising compilers. Some of these approaches [18, 4] make use of fractional factorial designs for experiment planning.

More recently a broader range of randomised search algorithms have found wider attention among compiler researchers. In particular, the works of Cooper et al. [6] and Triantafyllis et al. [21] are relevant in the context of this paper. [6] is probably most similar to our work and has its main focus on evaluating the effectiveness of various optimisation algorithms for the search of low-level compiler phase orders within a platform-specific native compiler. In [21] an algorithm for compiler optimisation space exploration in EPIC-type machines is presented. Similar to our approach, different optimisation configurations are applied to each code segment. The main differences to our work, however, are in the level, kind and number of transformations considered and the approach to execution time estimation. Our approach not only deals with a much larger optimisation space (16 (in [6]) or 15 (in [21]) vs 81 transformations), but also considers additional dimensions introduced by transformation parameters and outperforms their technique. By using highly portable source-to-source code and data restructuring techniques our transformation toolkit can already be employed successfully during early development stages of a native compiler and will continue to deliver performance benefits as this compiler matures. In contrast to [6] we do not estimate the actual execution time by counting instructions based on an abstract RISC machine, but employ real embedded hardware to measure cycle accurate execution time. This alleviates the inevitable difficulty (as shown in [21]) in predicting and estimating the possible performance impact on the highly specialised and often idiosyncratic architectures of most embedded processors. In contrast to [21] we do not rely on compiler writer supplied predictive heuristics and configuration pruning to handle large search spaces, but leave this decision to the employed search algorithm.

7. CONCLUSION

In this paper we have described a probabilistic optimisation algorithm for finding good source-level transformation sequences for typical embedded programs written in C. We have demonstrated that source-to-source transformations are not only highly portable, but provide a much larger scope for performance improvements than any other low-level technique. Two competing search strategies provide a good balance between optimisation space exploration and focused search in the neighbourhood of already identified good candidates. We have integrated both parameter-less global and parameterised local transformations in a unified optimisation framework that can efficiently operate on a huge optimisation space spanned by more than 80 transformations.

The empirical evaluation of our optimisation toolkit based on three real embedded architectures and kernels and applications from the UTDSP benchmark suite has successfully demonstrated that our approach is able to outperform any other existing approach and gives an average speedup of 1.71 across platforms. Future work will investigate the integration of machine learning techniques based on program features into our optimisation algorithm.

8. REFERENCES

[1] S. Baluja. Population-Based Incremental Learning: A Method for Integrating Genetic Search Based Function Optimization and Competitive Learning Source. Technical Report: CS-94-163, Carnegie Mellon University, Pittsburgh, PA, 1994

[2] F. Bodin, Z. Chamski, C. Eisenbeis, E. Rohou, and A. Seznec. GCDS: A compiler strategy for trading code size against performance in embedded applications. Technical Report RR-3346, INRIA, France, 1998.

[3] C. Brandolese, W. Fornaciari, F. Salice, D. Sciuto. Source-Level Execution Time Estimation of C Programs. In *Proceedings of the 9th International Workshop on Hardare/Software Co-Design (CODES'01)*, Copenhagen, Denmark, April 2001.

[4] K. Chow and Y. Wu. Feedback-directed selection and characterization of compiler optimizations. In *Proceedings of the 4th Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, December 2001.

[5] K. D. Cooper, D. Subramanian, and L. Torczon. Adaptive Optimizing Compilers for the 21st Century. In *Proceedings of the 2001 LACSI Symposium*, Los Alamos Computer Science Institute, October 2001.

[6] K. D. Cooper, A. Grosul, T.J. Harvey, S. Reeves, D. Subramanian, L. Torzon, and T. Waterman Exploring the Structure of the Space of Compilation Sequences Using Randomized Search Algorithms In *Proceedings of the 2004 LACSI Symposium*, Santa Fe, NM, October 2004.

[7] H. Falk. An approach for automated application of platform-dependent source code transformations. http://ls12-www.cs.uni-dortmund.de/~falk/, 2001.

[8] B. Franke and M. O'Boyle. Array recovery and high-level transformations for DSP applications. *ACM Transactions on Embedded Computing Systems (TECS)*, 2(2):132–162, May 2003.

[9] G. Fursin, M. O'Boyle, and P. Knijnenburg. Evaluating iterative compilation. In *Proceedings of Languages and Compilers for Parallel Computers (LCPC'02)*, College Park, MD, USA, 2002.

[10] E.F. Granston and A. Holler. Automatic recommendation of compiler options. In *Proceedings of the 4th Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, December 2001.

[11] M. Hall, L. Anderson, S. Amarasinghe, B. Murphy, S.W. Liao, E. Bugnion, M. and Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, **29**(12), 84–89, 1999

[12] M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and H. S. Kim. Experimental evaluation of energy behavior of iteration space tiling. In *Proceedings of the 13th International Workshop on Languages and Compilers for Parallel Computing (LCPC'00)*, pages 142–157, Yorktown Heights, NY, USA, 2000.

[13] T. Kisuki, P.M. Knijnenburg, and M.F. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques (PACT'00)*, pages 237–248, October 2000.

[14] P. Kulkarni, W. Zhao, H. Moon, K. Cho, D. Whalley, J. Davidson, M. Bailey, Y. Park, and K. Gallivan. Finding effective optimization phase sequences. In *Proceedings of the 2003 ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'03)*, pages 12–23, June 2003.

[15] C. Lee. UTDSP benchmark suite. http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html, 1998.

[16] S. Liao, S. Devadas, K. Keutzer, S.W.K. Tjiang and A. Wang. Code Optimization Techniques for Embedded DSP Microprocessors. In *Proceedings of the 1995 Design Automation Conference (DAC'95)*, pp. 599–604, 1995

[17] C. Liem, P. Paulin, and A. Jerraya. Address calculation for retargetable compilation and exploration of instruction-set architectures. In *Proceedings of 33rd ACM Design Automation Conference (DAC '96)*, pages 597–600, Las Vegas, NV, USA, 1996.

[18] R.P.J. Pinkers, P.M.W. Knijnenburg, M. Haneda and H.A.G. Wijshoff. Statistical Selection of Compiler Options. In *Proceedings of MASCOTS*, pp. 494-501, 2004.

[19] M. Saghir, P. Chow, and C. Lee. A comparison of traditional and VLIW DSP architecture for compiled DSP applications. In *International Workshop on Compiler and Architecture Support for Embedded Systems (CASES '98)*, Washington, DC, USA, 1998.

[20] B. Su, J. Wang, and A. Esguerra. Source-level loop optimization for DSP code generation. In *Proceedings of 1999 IEEE International Conference on Acoustic, Speech and Signal Processing (ICASSP '99)*, volume 4, pages 2155–2158, Phoenix, AZ, 1999.

[21] S. Triantafyllis, M. Vachharajani, and D.I. August. Compiler Optimization-Space Exploration. *The Journal of Instruction-Level Parallelism*, volume 7, January 2005.

[22] M.J. Voss and R. Eigenmann. High-level adaptive program optimization with ADAPT. *ACM SIGPLAN Notices*, 36(7):93–102, 2001.

[23] V. Zivojnovic, J. Velarde, C. Schlager, and H. Meyr. DSPstone: A DSP-oriented benchmarking methodology. In *Proceedings of the International Conference on Signal Processing Applications & Technology (ICSPAT '94)*, pages 715–720, Dallas, TX, USA, 1994.