Combining Source-to-Source Transformations and Processor Instruction Set Extensions for the Automated Design-Space Exploration of Embedded Systems

Richard Vincent Bennett Alastair Colin Murray Björn Franke Nigel Topham

University of Edinburgh
School of Informatics
Institute for Computing Systems Architecture (ICSA)
{r.v.bennett,a.c.murray}@sms.ed.ac.uk {bfranke,npt}@inf.ed.ac.uk

Abstract

Industry's demand for flexible embedded solutions providing high performance and short time-to-market has led to the development of configurable and extensible processors. These pre-verified application-specific processors build on proven baseline cores while allowing for some degree of customization through user-defined instruction set extensions (ISE) implemented as functional units in an extended micro-architecture. The traditional design flow for ISE is based on plain C sources of the target application and, after some ISE identification and synthesis stages, a modified source file is produced with explicit handles to the new machine instructions. Further code optimization is left to the compiler. In this paper we develop a novel approach, namely the combined exploration of source-level transformations and ISE identification. We have combined automated code transformation and ISE generators to explore the potential benefits of such a combination. This applies up to 50 transformations from a selection of 70, and synthesizes ISEs for the resulting code. The resulting performance has been measured on 26 applications from the SNU-RT and UTDSP benchmarks. We show that the instruction extensions generated by automated tools are heavily influenced by source code structure. Our results demonstrate that a combination of source-level transformations and instruction set extensions can yield average performance improvements of 47%. This outperforms instruction set extensions when applied in isolation, and in extreme cases yields a speedup of 2.85.

Categories and Subject Descriptors B.8 [*Performance and Reliability*]: Performance Analysis and Design Aids; C.3 [*Special-Purpose and Application-Based Systems*]: Real-time and embedded systems; D.3.4 [*Processors*]: Optimization

General Terms Design, Performance

Keywords Customizable Processors, ASIPs, Source-Level Transformations, Compilers, Instruction Set Extension, Design Space Exploration

LCTES'07 June 13–16, 2007, San Diego, California, USA.

1. Introduction

High performance and short time-to-market are two of the major factors in embedded systems design. The goal is to deliver the best performance for a given cost and with the shortest possible design time. In recent years, processor IP vendors have addressed these goals by developing configurable and extensible processors such as the ARC 600 and 700, Tensilica Xtensa, ARM OptimoDE, and the MIPS Pro series. These cores provide system designers with pre-verified solutions, thus reducing risk involved in and cost of a new processor design. Also offered are large degrees of flexibility through custom-specific instruction set extensions (ISE), which may help improve performance of compute-intensive kernels. As a result of this specialization an optimized application-specific processor is derived from a generic processor template.

In order to explore different ISEs during the design stage and to trade off various, partially contradictory design goals (e.g. performance, power, chip area) tool support is indispensable. Existing commercial (e.g. [1]) and academic (e.g. [2]) tools analyze an application written in C, identify candidate instruction templates, modify the application's source code and insert handles to the newly created instructions. In general, the overall effectiveness of this approach depends on the designer's ability to generate complex instruction templates that (a) can replace a sufficiently large number of simple machine instructions, (b) are frequently executed and (c) can be efficiently implemented. This paper addresses problems (a) and (b), and we show that the selection of "good" instruction templates is strongly dependent on the shape of the C code presented to ISE generation tool. We propose a novel methodology that combines the exploration of high-level program transformations and low-level instruction templates. Using a probabilistic search algorithm and a source-level transformation tool we generate many different – but semantically equivalent – versions of the input program. We present all of these generated programs to an integer linear programming (ILP) based ISE tool. For each program a set of new instructions is generated along with profiling based estimates of the execution time and code size resulting from the exploitation of the new instructions. Using such an approach we achieve remarkable performance improvements – on average a 1.47x speedup across 26 computationally intensive embedded applications. We demonstrate that our approach enables the generation of more powerful ISEs, unachievable by traditional techniques, with no additional ISE tool effort.

```
unsigned short icrc1(unsigned short crc,
                     unsigned char onech)
{
  int i;
  unsigned short ans = (crc^onech << 8);

  for (i=0;i<8;i++) {
    if (ans & 0x8000)
      ans = (ans <<= 1) ^ 4129;
    else
      ans <<= 1;
  }
  return ans;
}



unsigned short icrc(unsigned short crc,
                    unsigned long len,
                    short jinit, int jrev)
{
  /* Some variable declarations */
  /* ... */


  if (!init) {
    init=1;
    for (j=0;j<=255;j++)
    {
      icrctb[j]=icrc1(j << 8,(uchar)0);
      rchr[j]=(uchar)(it[j & 0xF] << 4 |
                      it[j >> 4]);
    }
  }

  if (jinit >= 0)
    cword=((uchar) jinit) |
      (((uchar) jinit) << 8);
  else
    if (jrev < 0)
      cword=rchr[HIBYTE(cword)] |
        rchr[LOBYTE(cword)] << 8;

  for (j=1;j<=len;j++)
  {
    if (jrev < 0)
      tmp1 = rchr[lin[j]]^ HIBYTE(cword);
    else
      tmp1 = lin[j]^ HIBYTE(cword);
    cword = icrctb[tmp1] ^ LOBYTE(cword) << 8;
  }

  if (jrev >= 0)
    tmp2 = cword;
  else
    tmp2 = rchr[HIBYTE(cword)] |
      rchr[LOBYTE(cword)] << 8;

  return (tmp2);
}
```

(a) Original SNU-RT CRC implementation

```
unsigned short icrc1(unsigned short crc, /*...*/)
{
  /*...*/

  /* Loop unrolling */
  for (i=0;i<8;i+=2) {
    if (ans & 0x8000) ans = (ans <<= 1) ^ 4129;
    else ans <<= 1;
    /*...*/
  }
  return ans;
}

unsigned short icrc(unsigned short crc, /*...*/)
{
  /*...*/

  /* Bit packing */
  suif_tmp = (cword /*...*/) |
    (jinit /*...*/) >> /*...*/ |
    (jrev & /*...*/) >> /*...*/ |
    (len & /*...*/) >> /*...*/;

  cword = 1u & suif_tmp;

  if (!init) {
    init = 1;
    /* Loop lowering */
    j = 0;
    do {
      /* Computation of icrctb & rchr ...*/
    } while (j <= 255);
  }
  /* Bit unpacking */
  if (0 <= ((2u & suif_tmp) >> 1u))
    cword = ((2u & suif_tmp) >> 1u) |
      (((2u & suif_tmp) >> 1u) << 8u);
  else
    if (((4u & suif_tmp) >> 2u) < 0)
      cword = /*...*/

  j = 1;
  if (1ul <= ((8u & suif_tmp) >> 3u)) {
    /* Move loop invariant conditionals */
    if (((4u & suif_tmp) >> 2u) < 0) {
      /* Loop lowering */
      do {
        /* Computation of cword ... */
      } while (j <= ((8u & suif_tmp) >> 3u));
    }
    else {
      /* Similar lowered loop as before ... */
    }
  }

  if (0 <= (4u & suif_tmp) >> 2u) tmp2 = cword;
  else tmp2 = /*...*/;

  return tmp2;
}
```

(b) CRC after transformation

Figure 1. Original *SNU-RT CRC* implementation (a) and after application of source-level transformations resulting in best combined performance (b).

1.1 Motivating Example

As an example, consider the code excerpt in figure 1(a). The two functions *icrc1* and *icrc* are part of the SNU-RT *CRC* benchmark and implement a cyclic redundancy check for an input string stored in the array *lin[]*. The key features of the code are small *for* loops in both functions, which contain conditional branches and perform a larger number of bit-level manipulation operations. Presented with this plain code current instruction set extension technology (see Section 4) generates new instruction templates, which result in a 25% performance improvement over the baseline code.

In figure 1(b) the main differences due to source-level transformations of the code in figure 1(a) are shown. While the code is functionally equivalent, it outperforms the code in figure 1(a) by a factor of 1.63. Loop unrolling has been applied to the *for* loop in the *icrc1* function. This reduces the loop overhead and improves flexibility for instruction scheduling. In the *icrc* function the effects of source-level transformation are more fundamental. *for* loops in the code have been lowered into *do-while* loops and, most important, *bit-packing* and *hoisting of loop invariant conditionals* transformations have been applied. *Bit-packing* packs multiple variables into a single variable of type integer and, on its own, usually degrades performance. When combined with ISE generation, however, otherwise expensive bit-level manipulation operations for packing and unpacking can be encoded as complex, but fast instructions and yield an overall performance improvement. In fact, the instruction templates generated for example 1(b) are generally more complex than those generated from the baseline code. An example of such an instruction is shown in figure 1 and it implements the mentioned packing/unpacking operation. Moving loop invariant conditional outside the loop eliminates redundant comparisons and jumps and further increases performance.

ISE generation based on the transformed code in figure 1(b) result in a further 23% improvement (over just transformed code), or a total combined speedup of 2.02x over the baseline. Only a certain part of the performance gain can be directly attributed to code transformations, the rest is due to the enabling effect of the source-level transformations on the ISE generation.

This short example demonstrates how difficult it is to predict the best source-level transformation and instruction set extension for a given application. It also shows that high-level code and low-level architecture optimization cannot be separated, but are tightly coupled. Combined exploration of both the software and hardware design spaces generate a significantly better solution than isolated optimization approaches could produce. In this paper we present an empirical evaluation of this HW/SW design space interaction and show that a probabilistic search algorithm is able to examine a small fraction of the optimization space and still find significant performance improvements.

The rest of the paper is organized as follows. In section 2 we discuss the large body of related work. Section 3 presents the background to our research and we discuss the problem of combined HW/SW design space exploration. In section 4 we describe our methodology and the experimental setup. Section 5 discusses our results, while section 6 concludes this paper.

2. Related Work

This paper seeks to broaden the understanding of the potential for known transformation techniques to improve the quality of automated instruction set selection. The current state of the art in *Instruction Set Extension* is described in section 2.1, followed by a description of the *Source-to-Source Transformations* in section 2.2. The large field of *HW/SW Codesign* is summarized in 2.3.

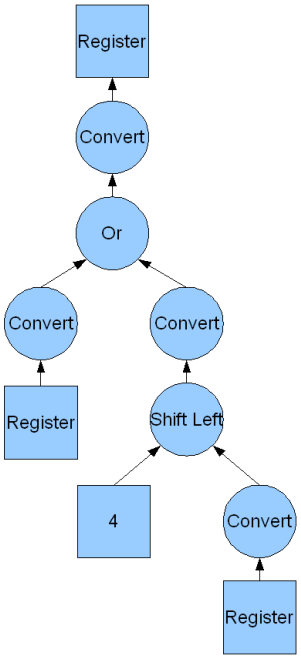


Figure 2. Complex instruction template generated for the transformed CRC code in figure 1(b).

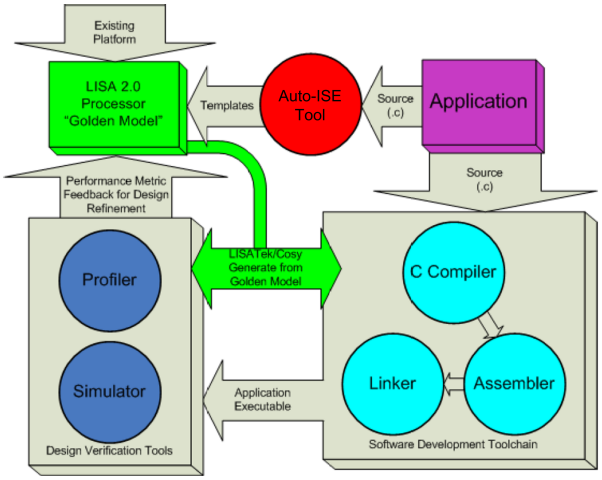


Figure 3. The Compiler-in-loop methodology for ASIP design space exploration.

2.1 Automated Instruction Set Extension

The automation of ISE exploration has been actively studied in recent years, leading to a number of algorithms [3, 4, 5, 6] which derive the partitioning of hardware and software under micro-architectural constraints. Work is still being formed in defining the full space of exploration even in purely arithmetic instruction set design [7]. Work to include better models in tools has allowed for better decisions about the performance and feasibility of extensions [8].

The current exploration approach of using a range of tools operating on a canonical system-level ADL, is described as "Compiler-

in-Loop Design-Space Exploration" [9]. It was originally motivated [10] through the discovery that iterative and methodical exploration of ASIP design is very beneficial in decreasing time-to-market. CoSy [11] and LISATek [12] tools feature in many such frameworks; Figure 3 illustrates such a combination.

Earlier efforts [13] to combine code transformation and ISE have been targeted at CDFG transformation towards a more efficient arithmetic structure. This operates post automated ISE (AISE), so does not directly contribute to the design space search but improves upon the result.

In [14] it is shown that an exploration of *if-conversion* and *loop-unrolling* source-to-source transformations is successful in enabling better performing AISE. We are motivated by this approach to perform the comprehensive and systematic study of this paper, as it demonstrates the effectiveness of ISE-targeted heuristics in transformation.

2.2 Source-to-Source Transformations

Due to their inherent portability and large scope source-level transformations have been used for various fields such as code optimizations targeting I/O performance [15] or energy efficiency [16, 17], formal verification [18], and, most notably, for single- and multi-core performance optimization of computationally intensive embedded applications (e.g. [19, 20, 21] and [22], respectively).

ROSE [23] and Transformers [24] are tools for building source-to-source transformation tools for the optimization of C and C++ object-oriented applications.

An empirical study of source-level transformations for digital signal processing applications is subject of [20], and more comprehensive studies in the context of machine-learning based adaptive compilation can be found in [25] and [26].

2.3 HW/SW Codesign

HW/SW Codesign was an active research area in the 90's and has inspired subsequent work on *Electronic System Level Design*. A comprehensive summary of research directions, approaches and tools can be found in e.g. [27]. This work covers a broad scope of issues, typically ranging from the analysis of constraints and requirements down to system evaluation and design verification. In contrast, our work focuses on a more specific, individual problem, namely that of HW/SW partitioning in the context of extensible application-specific processors.

3. Background

3.1 Extensible Processors

Extensible processors contain a number of variable components, essentially opening up design spaces inside the processor core for exploration by the designer of an ASIP-based system. Extensions to registers and supporting arithmetic logic are implemented outside of a prefab baseline core, the latter implementing all of the expected basic RISC functionality. In this manner users may make the best use of the degrees of freedom provided, with the knowledge that their extensions will not make unpredictable timing changes to the core as a whole. Some cores allow for reconfiguration through FPGA fabrics, whereas others are provided as well-documented IP-blocks. Either is extended through implementing the extensions in SystemC or Verilog with respect to the architecture's extension interface.

3.2 Design-Space Exploration

Design-Space Exploration (DSE) is an optimization process in the design flow of a System-on-Chip (SoC), typically operating under multiple constraints (performance, power & energy, cost & complexity etc.), and targeting an often multi-dimensional and highly

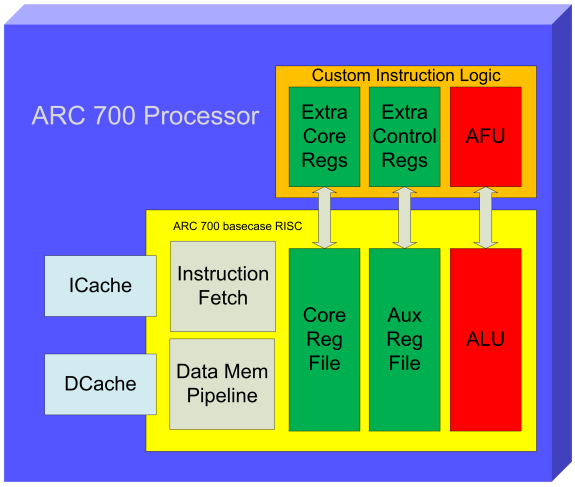


Figure 4. A simplified system-level view of ARC700 family architecture, demonstrating the pre-verified baseline core and its connection to an instruction set extension through custom registers and arithmetic units.

non-linear optimization space. Multiple dependent levels (algorithm, SW, and HW design space exploration) of interaction make it difficult to employ isolated local search approaches, but require a combined effort crossing the traditional boundaries between design domains, providing feedback paths and integrating tools into larger frameworks.

Configurable and extensible processor cores such as the ARC 600 and 700 have a number of capabilities to allow their instruction set and micro-architecture to be optimized for a particular application. Design concerns guiding DSE are often reduced to metrics such as execution speed, power usage, and die area; each of these metrics has an accompanying relevant design space in the configuration and extension domain.

Meeting this requirement means that no further increase in speed is generally useful, other than to provide an overhead for development. Application deadlines will be met and the system built around the core will be able to communicate and process data without stalling due to system-level deadlines missed by the core.

Once execution speed requirements have been met the focus of designers may be switched to secondary axes of design concern, such as power usage. Efforts in addressing one axis of design concern may make use of excesses in other axes. For example if performance exceeds requirements the clock speed of the ASIP may be reduced, reducing the power consumption. These secondary concerns have additional design spaces of the configurable core available to be explored for satisfactory areas; for example clock gating, dynamic voltage scaling, and unit pruning. These are however outside the scope of instruction set extension and not covered here.

Unfortunately the "second order" effects of core extension are not always beneficial and often hard to predict with any accuracy. Adding more logic to a core can for example increase the critical path and force a reduction in the overall clock speed. Such a complicated web of non-orthogonal trade-offs forms a space which can only be explored efficiently through the aid of iterative automated means.

Instruction set extensions affect all three of the mentioned axes of design concern. The guiding metric in deriving extensions is often still application execution speed; designers will add instructions that "cover" the hottest (most frequently executed) sections of their application code. The intention is that by partitioning of the application code into areas covered and not by ISE, subsections

of micro-architecture can be dedicated to the servicing of these new instructions. In this highly application-specific design space, several sources of micro-architectural optimization are currently brought to bear on the hardware performance of the new instruction:

- Operation-level/Spatial parallelism; parallel instances of arithmetic hardware in order to perform multiple operations at-once, as allowed by dependencies.
- Reduced register-transfer overhead, due to the increased locality of communication within the functional unit used to represent the new instruction.
- Aggregation of clock period surplus present in most arithmetic functions. In particular, bitwise functions have a hardware latency far below the clock period in most cases.

This growing catalog of optimization aims to ensure that the "hot-spot" represented by the new instruction achieves the maximum speed possible, by trading off die area for an increase in execution speed, a decrease in power usage, and a decrease in code-size. Often these extensions correlate to very frequently executed sections of code, and so the benefits for a relatively small increase in die-size can be very tempting to designers. The problem remains to find a way to accurately model both the existing architecture and the full range of potential extensions in such a way as to efficiently automate exploration.

It has been shown [14] that new search methods and heuristics can be developed to control the application of transformations, with respect to the new set of goals inherent in ISE as compared to code generation. Transformations once targeted at the back-end would attempt to limit increasing basic block size due to register pressure. Now in instruction set extension the drive is towards the largest possible basic block size for analysis.

3.3 Combined Design Space

The combined design space in question here is that of transformation and ISE, with the intention of demonstrating that there is promise for automated techniques to manage the design in such a large space. Also important, that the results of a cooperative automated framework can outweigh the sum of their separated components.

The hope is, as with compilers, that the actual efforts of the search of the combined space can remain a phased searching of each space individually. The most important factors in this scenario are the accuracy and detail of the modeling employed in any decision making. This work attempts to contribute to the understanding of which transformations will need to be made extension-aware, and which are invariantly beneficial under extension.

The use of compiler transforms when developing automated design space exploration must be very carefully considered, so as not to disturb the context in which design-space decisions are made. Transformations which are run prior to an automated ISE tool must be re-applied with the same parameters to the areas in which the tool identified mappings. Otherwise, the ISE will not find the same mapping in code-generation without having the areas explicitly defined by manual means.

4. Experiment Methodology

The primary concern of our experiment was to determine which transformations or combinations thereof infer the greatest execution speed improvement to application-specific software under ISE automation. Secondly, the experiment was to find limits for performance gain and loss from the combined design space defined by transformation and ISE over a baseline design employing neither. With this information, we are well equipped to properly focus the efforts of future research towards the most beneficial transformations for ISE.

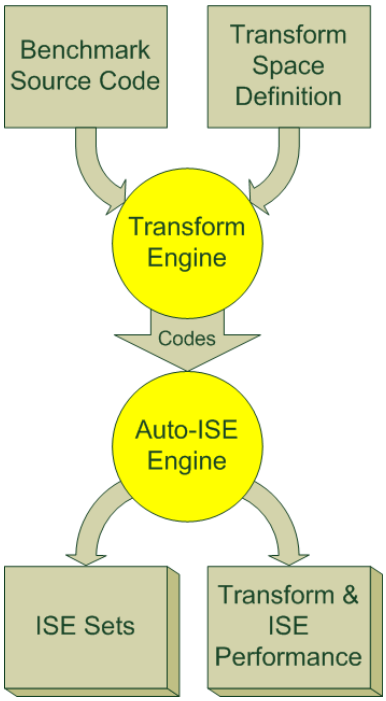


Figure 5. The combined but phased searching of transform and ISE design spaces; our experiment methodology as a flow diagram.

To represent the transformation design space in this experiment, we use a source to source transformation tool built upon the SUIF1 [28] compiler framework. Samples are taken with uniform probability, at random point across the entire space of potential transformation. A sample in this sense represents a single point in the transformation space, and results in the ordered set of transformations selected at that sample point to be applied to the code. The tool generates large volumes of transformed source code samples rapidly from a definition of:

- The source code, in C; for our purposes a variety of single-function benchmarks are tested.
- The Transform Space Definition, as the boolean inclusion or exclusion of transforms permitted in the space, plus the maximum number of transformation phases for each sample. The tool supports a wide array of source to source transformations to be used in the exploration [25]. We allowed a maximum of 50 phases in our samples.
- The number of samples to take from the transformation space, and hence the number of transformed source codes to produce. In our case this was set to 10,000, however some sequences of transformation produced invalid code and hence were culled from the results. The number of valid transformation sequences is covered in Section 5.

The benchmarks used in this experiment were taken from SNU-RT [29] and UTDSP [30] suites. Those taken were as follows:

- **SNU-RT**; adpcm, crc, fft1, fft1k, fir, jfdctint, lms, ludcmp, matmul, minver, qsort-exam, qurt, select.

- **UTDSP**; edge_detect, fft_1024, fft_256, fir_256, fir_32_1, histogram, iir_4_64, latnrm_32_64, latnrm_8_1, lmsfir_32_64, lmsfir_8_1, mult_10_10, mult_4_4.

We store for each benchmark the entire set of transformed source codes representing that benchmark after sample points in the transformation space are applied to it. The set of transforms applied at each sample is also stored for later correlation in analysis.

This set of transformed source codes forms a representative sampling of the entire search space for that benchmark, each sample is then processed by an automated profiling ISE tool based on the Atasu et al. Integer Linear Programming method of derivation [5]. The tool operates in three phases:

- Instrumentation; wherein the ISE tool augments the intermediate representation of the application with counters for profiling. The CoSy-based tool emits the i686 assembly for this profiling executable which is then assembled and run using the standard GNU tool chain.
- Execution; running the instrumented binary records per-basic-block execution frequencies, which are stored in a file for use by the extension phase.
- Extension; The IR is augmented with profiling statistics, which are then used to select the top four instructions using the Atasu ILP AISE algorithm [5]. The ISE tool's profiler combined with a latency table for the given target architecture produces runtime and code-size performance metrics for the original transform-space sample. These metrics and the generated instructions are stored alongside the transformed code and transform-point definition.

4.1 Algorithms

The algorithm controlling the source-level transformation of the input program is a probabilistic algorithm derived from [25]. Combining a space exploration component with machine-learning focused search this algorithm maintains probability vector representing the selection probability of each transformation. Starting with a uniform probability distribution the selection probabilities are updated after each iteration based on the speedup of the transformed program.

The ILP AISE algorithm used generates data-flow-graph templates through conversion of basic blocks to a set of constraints in an *Integer Linear Program (ILP)* and solution of that program. For our implementation a tool built into a CoSy compiler uses the *lp_solve* library to solve such problems and generate a set of candidate templates for an entire program. Constraints are declared from each basic block to generate a template such that:

- The template is convex (i.e. does not have any holes), so that it may be scheduled.
- Input and output port constraints are met (i.e. the number of register input and output ports are sufficient), so that it may be implemented.

In addition to the constraints, a goal function is also expressed. For this algorithm the goal is the estimated serial time of execution in cycles of the instructions covered by the template, minus the estimated critical path of the template. The former is denoted the "Software" execution time, and is indicative of the time the instruction would take to execute on the unextended architecture. The latter is denoted the "Hardware" execution time, and is a real-valued factor of the cycle time taken to execute the template as a single instruction. A cycle time for each Software and Hardware operation is specified to the tool a-priori to ILP construction, to allow for the constraints to be generated. The per-template difference between Software and Hardware execution time is the per-execution gain in cycles to an architecture implementing that template. Following the generation of templates from basic blocks, the templates are checked for isomorphism with one another using the *NAUTY* graph isomorphism library, then ranked using the product of their estimated usage and per-execution gain. The top four of these instructions are then recorded alongside their performance estimates for inclusion in results.

4.2 Experimental Setup

For the purposes of this experiment, we configured our latencies to those of an Intel XScale PXA270 processor, a current high-performance embedded micro-architecture based upon the ARM7 instruction set. An input/output port constraint of 8/8 is set, to allow a wide range of potential ISE and avoid limitations on our results due to the synthetic micro-architectural constraints set in the ISE algorithm. It has been shown [8] that pipelining of ISE extensions is possible to reduce per-cycle register file I/O to suit actual requirements.

We therefore have for each benchmark, for each of up to 10,000 transformation-space sample points:

- Source code after transformation.
- Instruction Set Extensions defined as data-flow templates.
- A record of performance in cycles (runtime) and instructions (code-size) before and after the transformations are applied to the benchmark.
- A likewise record of the improvement to each of the performance metrics for each of the instructions generated by automated ISE for the transformed source.
- Aggregation of the results of the top four of these instructions to calculate the overall benefit to the transformed code.

So that there is a control point for reference, we ensure that a baseline utilizing no transforms is sampled from the transformation space. Our tool produces as many ISE templates as it can find within the source code. However, we limit the number used in our results to four, in order to allow only the inclusion of the largest and best performing ISE's such as we expect to reveal through transformation. Current commercial approaches such as the Tensilica XPRES [31] tend to use large numbers of small instructions to preserve generality; our work assumes a very application-specific core is desired.

This entire experiment was run on a quad-core machine running Linux 2.6, over the course of several days in order to allow for the large-scale sampling. Our tools are "pipelined" in their operation to speed up results generation, as illustrated in Figure 5.

With these results recorded, we go on to make observations on the correlation of transformation and ISE performance, in how the spaces combine to form the more relevant performance measure: overall performance.

5. Results

Figures 6 and 7 show the runtime improvements achieved on a selection of benchmarks from the SNU-RT and UTDSP suites. For each benchmark the three bars represent the best improvement seen in the search space for each technique. Peak runtime improvements of 2.70x (SNU-RT ludcmp), 1.46x and 2.85x (both SNU-RT FFT) are seen, for transformations alone, ISE alone and the combination of the two, respectively. Average runtime improvements across both benchmark suites are 1.35x, 1.09x and 1.47x respectively. It can also be seen that of the 26 benchmarks considered 5 of them see a combined transformation and ISE runtime performance improvement of over 2.0x and only 6 see an improvement of less than 1.15x.

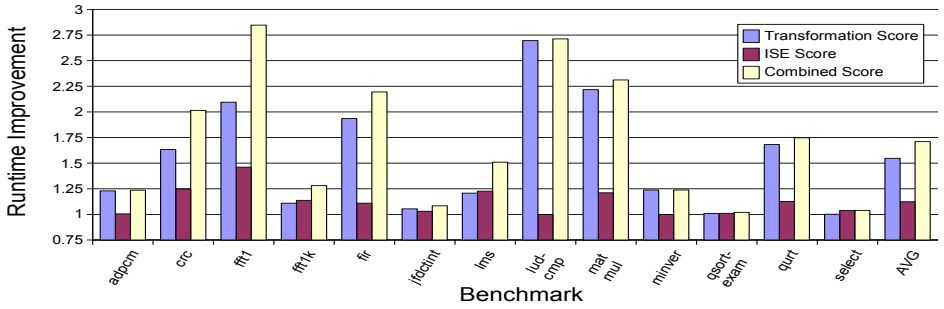


Figure 6. Runtime Improvements achieved on the SNU-RT benchmarks.

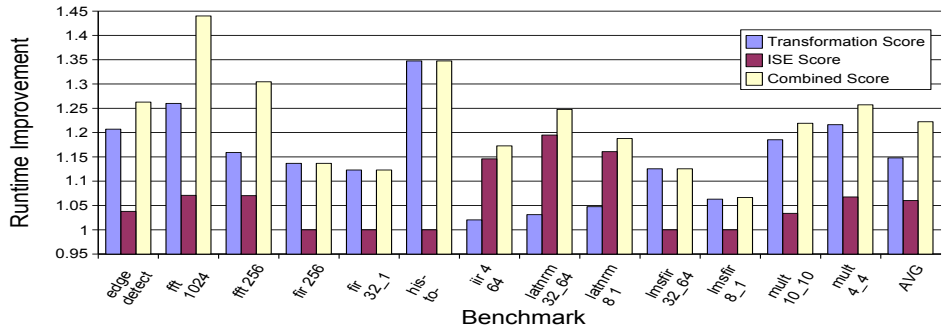


Figure 7. Runtime Improvements achieved on the UTDSP benchmarks.

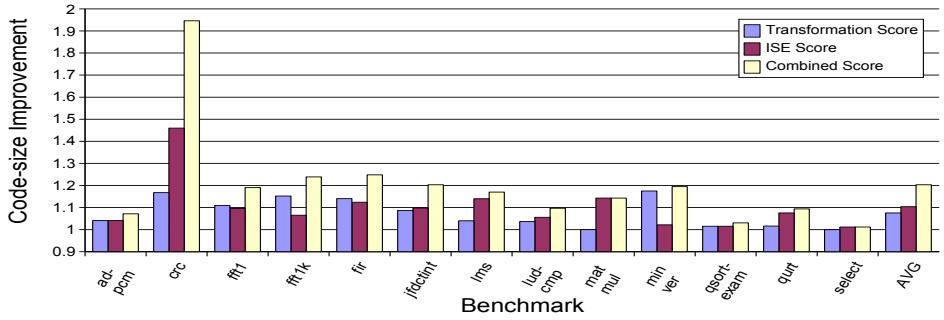


Figure 8. Code-size Improvements achieved on the SNU-RT benchmarks.

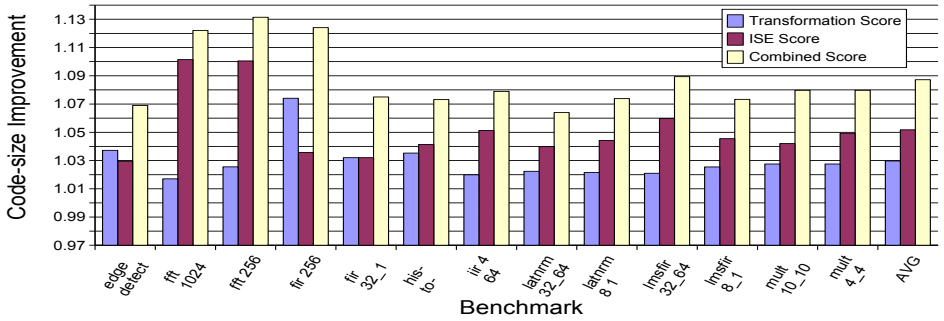
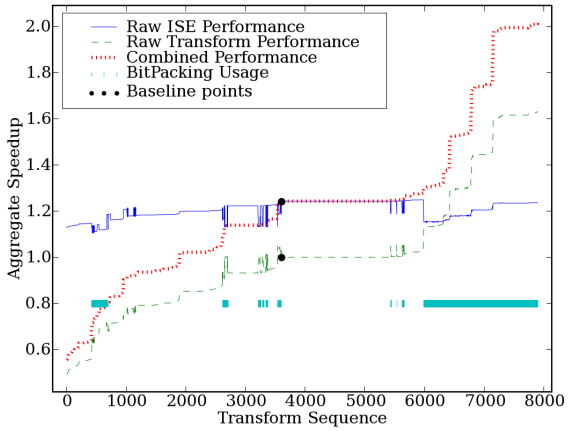
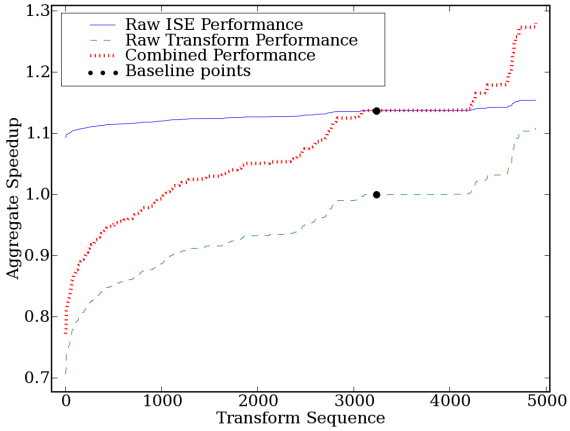


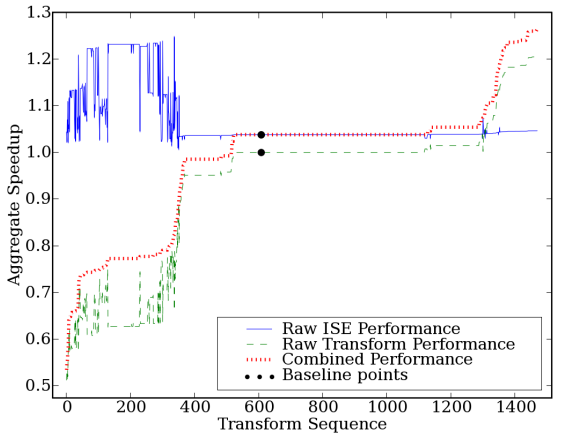
Figure 9. Code-size Improvements achieved on the UTDSP benchmarks.



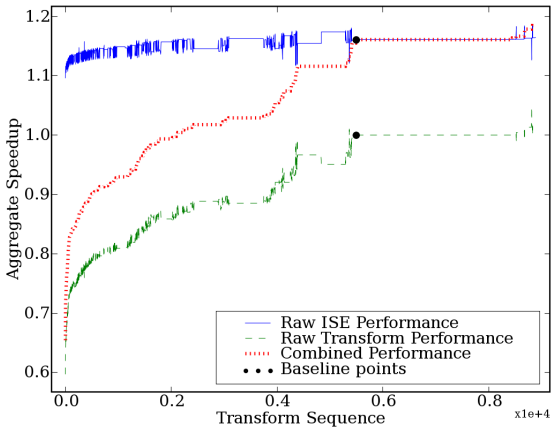
(a) SNU-RT CRC (based on 7896 runs)



(b) SNU-RT FFT1k (based on 4892 runs)



(c) UTDSP Edge Detect (based on 1470 runs)



(d) UTDSP Latnrm 8_1 (based on 8882 runs)

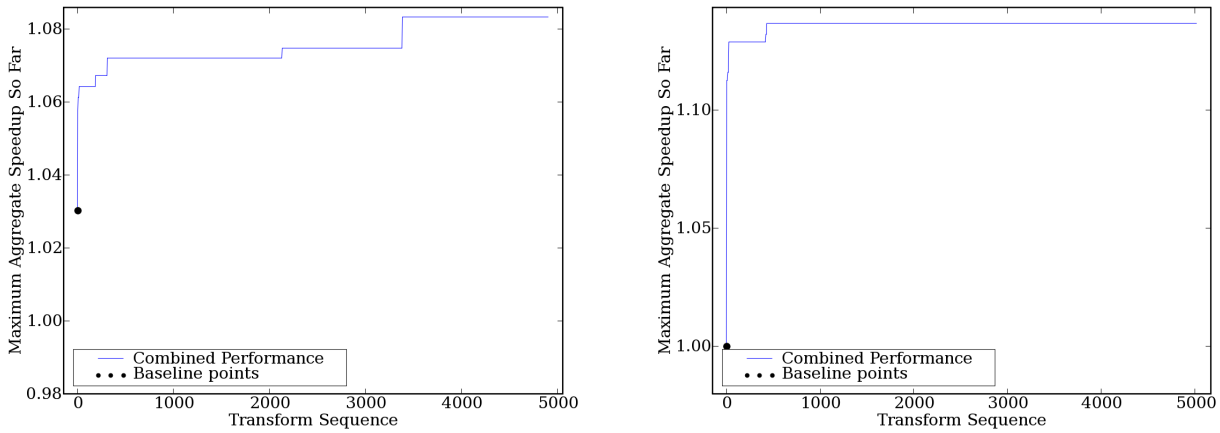
Figure 10. Runtime improvements achieved for every transformation sequence in the search space for a selection of benchmarks. For each version of the program (x-axis) three speedup values are shown: speedup after ISE only (raw ISE performance), speedup after source-level transformation only (raw transform performance), and speedup after combined ISE and source-level transformation (combined performance). The baseline points are the performance of each technique on unmodified code. The code versions are ordered by increasing combined performance along the x-axis.

Figures 8 and 9 show the code-size improvements achieved on the same benchmarks. These graphs are not based on the same sample points that runtime improvement figures are, but separate transformation sequences that were found to be effective at reducing code-size. Peak code-size improvements of 1.18x (SNU-RT minver), 1.46x and 1.95x (both SNU-RT CRC) are seen, for transformations alone, ISE alone and the combination of the two, respectively. Average code-size improvements across both benchmark suites are 1.05x, 1.08x and 1.15x respectively.

It can be seen that in figures 6, 7, 8 and 9 that the average results for the SNU-RT benchmarks are noticeably higher than for UTDSP. The primary reason for this is that the SNU-RT benchmarks are smaller, so the potential selection space is smaller and thus better suited to uniform sampling. Although we only explore a tiny fraction of the overall search space we still obtain very good results, however it seems likely that exploring a larger portion of the search space will yield even better results, especially for larger programs. Larger programs are also likely to benefit from a more directed search technique that can quickly focus on the promising areas on the search space, such as the one described in [25].

This shows that in many cases simply choosing transformations that allow effective use of ISE will not give good overall performance. Strong examples of this are the UTDSP FIR-256 and FIR-32_1 benchmarks, where the optimal combined performance is given by a set of transformations that did not allow any runtime improvement through ISE at all. Examples where combined performance is significantly better than either transformations or ISE alone are SNU-RT CRC and the SNU-RT FFT benchmark.

The graphs in figure 10 show the performance for each individual technique and the combination of the two for every sample point in the search space, for a small selection of benchmarks. The samples are sorted by the performance of combining transformations and ISE. This allows the ratio of transformation to ISE performance to be seen and also shows where there are correlations between the performance of the two individual techniques. These correlations are seen where either the performance of both individ-



(a) SNU-RT jfdctint

(b) UTDSP FIR-256

Figure 11. The maximum performance found so far for each point in the sample space and all points that were evaluated before it, ie. the development of the best performing program so far over the program versions generated by the source-level transformation tool.

ual techniques improve at the same point or where one gets better but the other gets worse.

An example of this correlation can be seen on the left side of figure 10(c) which shows the separated performance for sets of transformations which allow good ISE performance but perform poorly overall due to the performance decrease seen with transforms alone. A more useful example of the correlation between transformations and ISE is shown in our motivating example, SNU-RT CRC, with the bit packing transformation. Sequences that make use of this transform are marked as short vertical bars in figure 10(a). It can be seen that all the best performing sequences make use of this transformation. At the points where it changes from being turned off to turned on there is dip in ISE performance and a rise of about the same magnitude in transformation performance. It can be seen that transformation performance improves greatly (from negligible improvement without it to up to 1.63x in a sequence with it turned on) and ISE performance recovers. So with the correct surrounding transformations the bit packing transformation allows both code performance on its own and good ISE performance for this benchmark.

Figure 10(b) shows an almost ideal set of results (for SNU-RT FFT1k), where the best set of transformation sequences when considered alone also allow the most gain from ISE. When the optimal sequences overlap in this way the combined performance is very high (e.g. going from peaks of 1.11x and 1.14x with individual techniques to a peak of 1.28x with combined techniques for SNU-RT FFT1k.) Not all sequences give such clean results though, figure 10(a) (for our motivating example, SNU-RT CRC) shows patterns that are visible in several benchmarks. Specifically, the best performance is given by finding a transformation sequence that both improves the runtime on the code while not damaging the ISE potential. The optimal sequence actually gives ISE performance slightly below that which was achieved on the baseline code, but overall performance is high due to good runtime improvements from the transforms themselves. Figure 10(d) show the results from a benchmark where almost none of the overall improvement comes from transformations but almost entirely from ISE (UTDSP latnrm-8_1). However, the graph still shows that poor-code shape can limit ISE.

Figure 11 shows the performance of the best transformation sequence found so far as each point in the sample space is evaluated. Figure 11(a) shows an example (for the SNU-RT jfdctint benchmark) that has the kind of characteristics that led to evaluating such a high number of samples in the transformation space. It contains several steps in the performance of the best sequence found so far, with the very best not being found until after several thousand samples were evaluated. However, this was not typical of most benchmarks, figure 11(b) is an example (for UTDSP FIR-256) that shows the typical behavior. It also has steps in the performance of the best sequence found so far, but they are much closer together and the very best is found in about five hundred runs, with none of remaining sequences evaluated doing better. This suggests that considering a smaller number of samples should still give acceptable results, though considering a larger number of samples may find more steps leading to even greater performance.

Regarding the number of samples considered in the results it can be seen in figures 10 and 11 that the number varies between benchmarks. This is because some of transformation sequences either caused internal correctness checks in the SUIF compiler to be triggered, or in a few cases generated incorrect code. These samples are discarded, but there are still a great many remaining, these are used to generate the results here.

6. Conclusions

In this paper we have described a methodology for improved ISE generation that combines the exploration of high-level source transformations and low-level ISE identification. We have demonstrated that source-to-source transformations are not only very effective on their own, but provide much larger scope for performance improvement through ISE generation than any other isolated low-level technique. We have integrated both source-level transformations and ISE generation in a unified framework that can efficiently optimize both hardware and software design spaces for extensible processors.

The empirical evaluation of our design space exploration framework is based on a model of the Intel XScale processor and compute-intensive kernels and applications from the SNU-RT and UTDSP benchmark suites. We have successfully demonstrated that our approach is able to outperform any other existing approach and gives an average speedup of 1.47x. Compared to previous work [14], we have covered a much broader array of existing transfor-

mations to get a more global picture of the potential for transformation in improving instruction set extension. In addition, we have empirically demonstrated that there exists a non-trivial dependence between high-level transformations and the generated instruction set extensions justifying the co-exploration of the HW and SW design spaces.

Future work will investigate the integration of machine learning techniques based on program features into our design space exploration algorithm and target a commercial extensible processor platform.

References

[1] ARC International. ARChitect product brief, 2007.

[2] R. Leupers, K. Karuri, S. Kraemer, and M. Pandey. A design flow for configurable embedded processors based on optimized instruction set extension synthesis. In *Proceedings of Design Automation & Test in Europe (DATE)*, Munich, Germany, 2006.

[3] Armita Peymandoust, Laura Pozzi, Paolo Ienne, and Giovanni De Micheli. Automatic instruction set extension and utilisation for embedded processors. In *Proceedings of the 14th International Conference on Application-specific Systems, Architectures and Processors, The Hague, The Netherlands.*, 2003.

[4] Partha Biswas, Sundarshan Banerjee, Nikil D. Dutt, Laura Pozzi, and Paolo Ienne. ISEGEN: An iterative improvement-based ISE generation technique for fast customization of processors. *IEEE Transactions on VLSI*, 14(7), 2006.

[5] Kubilay Atasu, Gunhan Dundar, and Can Ozturan. An integer linear programming approach for identifying instruction-set extensions. In *Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ISSS '05)*, 2005.

[6] Laura Pozzi, Kubilay Atasu, and Paolo Ienne. Exact and approximate algorithms for the extension of embedded processor instruction sets. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(7):1209–1229, 2006.

[7] Ajay K. Verma and Paolo Ienne. Towards the automatic exploration of arithmetic circuit architectures. In *In Proceedings of the 43rd Design Automation Conference, San Francisco, California*, 2006.

[8] Laura Pozzi and Paolo Ienne. Exploiting pipelining to relax register-file port constraints of instruction-set extensions. In *In Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems, San Francisco, Calif*, pages 2–10, 2005.

[9] M. Hohenauer, H. Scharwaechter, K. Karuri, O. Wahlen, T. Kogel, R. Leupers, G. Ascheid, and H. Meyr. Compiler-in-loop architecture exploration for efficient application specific embedded processor design. In *Design & Elektronik*, Munich, Germany, WEKA Verlag, 2004.

[10] T. Glokler, A. Hoffmann, and H. Meyr. Methodical low-power ASIP design space exploration. *VLSI Signal Processing*, 33, 2003.

[11] *ACE CoSy Website - http://www.ace.nl/compiler/cosy.html*.

[12] *CoWare LISATek Datasheet - http://www.coware.com/PDF/products/LISATek.pdf*.

[13] Paolo Ienne and Ajay K. Verma. Arithmetic transformations to maximise the use of compressor trees. In *Proceedings of the IEEE International Workshop on Electronic Design, Test and Applications, Perth, Australia*, 2004.

[14] Paolo Bonzini and Laura Pozzi. Code transformation strategies for extensible embedded processors. In *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 242–252, New York, NY, USA, 2006. ACM Press.

[15] Yijian Wang and David Kaeli. Source level transformations to improve I/O data partitioning. In *Proceedings of the International Workshop on Storage Network Architecture and Parallel I/Os*, 2003.

[16] E. Chung, L. Benini, and G. De Micheli. Energy efficient source code transformation based on value profiling. In *Proceedings of the International Workshop on Compilers and Operating Systems for Low Power*, Philadelphia, USA, 2000.

[17] C. Kulkarni, F. Catthoor, and H. De Man. Code transformations for low power caching in embedded multimedia processors. In *Proceedings of the 12th. International Parallel Processing Symposium on International Parallel Processing Symposium*, pages 292–297, 1998.

[18] B.D. Winters and A.J. Hu. Source-level transformations for improved formal verification. In *Proceedings of the IEEE International Conference on Computer Design*, 2000.

[19] Heiko Falk and Peter Marwedel. *Source Code Optimization Techniques for Data Flow Dominated Embedded Software*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 2004.

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

[21] Victor De La Luz and Mahmut Kandemir. Array regrouping and its use in compiling data-intensive embedded applications. *IEEE Transactions on Computers*, 53(1):1–19, 2004.

[22] Björn Franke and Michael O'Boyle. Combining program recovery, auto-parallelisation and locality analysis for C programs on multi-processor embedded systems. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT'03)*, New Orleans, September/October 2003.

[23] Markus Schordan and Daniel J. Quinlan. A source-to-source architecture for user-defined optimizations. In *Proceedings of the Joint Modular Languages Conference*, 2003.

[24] Alexandre Borghi, Valentin David, and Akim Demaille. C-Transformers - a framework to write C program transformations. *ACM Crossroads*, 2004.

[25] Björn Franke, Michael O'Boyle, John Thomson, and Grigori Fursin. Probabilistic source-level optimisation of embedded programs. In *Proceedings of the 2005 Conference on Languages, Compilers and Tools for Embedded Systems (LCTES'05)*, 2005.

[26] Felix Agakov, Edwin Bonilla, John Cavazos, Björn Franke, Michael F.P. O'Boyle, John Thomson, Marc Toussaint, and Christopher K.I. Williams. Using machine learning to focus iterative optimization. In *Proceedings of the 4th Annual International Symposium on Code Generation and Optimization (CGO)*, 2006.

[27] Jerzy Rozenblit and Klaus Buchenrieder. *Codesign - Computer-Aided Software/Hardware Engineering*. IEEE Press, New York, 1995.

[28] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve W. K. Tjiang, Shih-Weui Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Notices*, 29(12), 1994.

[29] *SNU-RT Real-Time Benchmarks - http://archi.snu.ac.kr/realtime/benchmark/*.

[30] Corinna G. Lee. *UTDSP Benchmarks - http://www.eecg.toronto.edu/ corinna/DSP/infrastructure/UTDSP.html*, 1998.

[31] Tensilica Inc. The XPRES compiler: Triple-threat solution to code performance challenges. *Tensilica Inc Whitepaper*, 2005.