Automated Floating-point to Fixed-point Conversion with the *fixify* Environment

Pavle Belanović and Markus Rupp
CD Laboratory for Design Methodology of Signal Processing Algorithms
Vienna University of Technology, Austria
{pbelanov,mrupp}@nt.tuwien.ac.at

Abstract

Conversion from floating-point to fixed-point formats is a necessary step in the design process of embedded systems and is traditionally performed manually. Automating this conversion process brings significant and much needed improvement in the efficiency of the design process. The fixify environment presented here fully automates the conversion process and comprises three optimization methods. The restricted-set full search algorithm is suited to designs that will be implemented on DSP cores and is, for such designs, guaranteed to find globally optimal solutions. On the other hand, the greedy search algorithm finds solution in the continuous search space and produces nearly optimal results, with the shortest required runtime. The branch-and-bound algorithm also works in the continuous search space and finds optimal solutions, but requires relatively long runtimes.

1. Introduction

Design of modern embedded systems experiences increasing pressures, both in terms of the rapidly increasing system performance and tighter power consumption constraints [12], but also from the aspect of the ever-shortening time to market and competitive cost constraints [10]. Hence, the efficiency of the design process becomes increasingly critical.

Experience has shown that traditional design processes are failing to keep up with the pace of developments described above [3]. The numerous fragmentations of the design process, such as lack of interoperability between Electronic Design Automation (EDA) tool chains, poor design reuse, and hindered communications among the design teams working on the various design abstraction levels, are the most common factors behind this inefficiency [10].

Hence, a number of new design environments have recently been proposed to provide the necessary efficiency gains in the design process [4, 8]. One of these novel design environments is the Open Tool Integration Environment (OTIE).

This work has been funded by the Christian Doppler Laboratory for Design Methodology of Signal Processing Algorithms.

1.1. Open Tool Integration Environment

The OTIE has been proposed and implemented with the aim of providing an efficient, robust and consistent development environment for embedded systems [8]. One of the crucial features of this environment is its inherent **expandability**, largely due to the fact that OTIE is built around the concept of the Single System Description (SSD), where all information about the design, as well as all the refinement steps it has gone through and the design decisions taken along the way, are kept in one central repository - the SSD. Various commercial, academic, or even in-house-developed tools or entire tool chains can systematically be integrated into the OTIE, thus expanding its capabilities.

Due to its expandability, the OTIE is in a constant state of growth and improvement and has successfully been extended to cover large portions of the design flow, most notably in virtual prototyping [9] and automated hardware/software partitioning [1].

1.2. Floating-point to Fixed-point Conversion

Design of embedded systems typically starts at the so-called algorithmic level, where the initial concept of the system is turned into an executable model and high-level specifications of the system are verified. Development at this stage of the design process is aided by such tools as Matlab/Simulink, CoWare SPW, or Synopsys CoCentric System Studio.

In terms of the numeric formats used, algorithmic-level models invariably use floating-point formats, for several reasons. Firstly, while the algorithm itself is undergoing changes, it is necessary to disburden the designer from having to take numeric effects into consideration. Hence, using floating-point formats, the designer is free to modify the algorithm itself, without taking into consideration overflow and quantization effects. Also, floating-point formats are highly suitable for algorithmic modeling because they are natively supported on PC or workstation platforms, where algorithmic modeling usually takes place.

However, at the end of the design process lies the implementation stage, where all hardware and software components of the system are fully implemented in the chosen target technologies and then integrated to create the final product. At this design stage, various compilers, debuggers, and place-an-route tools are

the EDA tools most commonly used. Both the software and hardware components of the system at this stage use only fixed-point numeric formats, because this allows drastic savings in all traditional cost metrics: the required silicon area, power consumption and latency/throughput (i.e. performance) of the final implementation.

Thus, during the design process it is necessary to perform the conversion from floating-point to suitable fixed-point numeric formats, for all data channels in the system. This transition necessitates careful consideration of the ranges and precision required for each channel, the overflow and quantization effects created by the introduction of the fixed-point formats, and possible instability or non-convergence effects these may introduce. A trade-off optimization is hence formed, between minimizing introduced quantization noise (i.e. keeping the Signal-to-Quantization Noise Ratio - SQNR - above an acceptable threshold limit) and minimizing the overall bitwidths in the system, so as to minimize the total system implementation cost.

This trade-off is traditionally performed manually, with the designer estimating the effects of fixed-point formats through system simulation and determining the required bitwidths and rounding/overflow modes through previous experience or existing knowledge of the system architecture (such as predetermined bus or memory interface bitwidths). This iterative procedure is very time-consuming and can sometimes account for up to 50% of the total design effort [5]. Hence, a number of approaches to automating the conversion from floating-point to fixed-point formats has been proposed.

1.3. Related Work

Existing approaches to automating the floating-point to fixed-point conversion can be clearly divided into the analytical (or static) and statistical (or dynamic) approaches. Each group has a number of advantages and limitations.

Analytical approaches [5, 2] are based on thorough analysis of the algorithmic description and the underlying control-flow and data-flow graphs. These approaches typically establish relationships between bitwidth requirements of all the channels in the design, and can hence guarantee the validity of their results independent of the data processed by the design. Also, analytical approaches are fast, since they require no simulations of the system model.

However, analytical approaches suffer from a serious drawback of producing extremely conservative results, which are in general far inferior to those produced by the designer manually. Hence, these approaches are only useful in providing a starting point and indicative guidance for manual optimization.

Statistical approaches [6, 11] on the other hand, are capable of producing the same quality of results as the designer's manual effort. These approaches are dynamic because they require simulations of the system model and the analysis of its numeric performance. Hence, they need significantly more time to optimise the fixed-point formats in the design than analytical approaches, but are still much faster in doing so than the designer alone. Also, unlike analytical approaches, dynamic approaches are capable of fine control over the performance/cost trade-off, thus completely disburdening the designer and fully automating the conversion process.

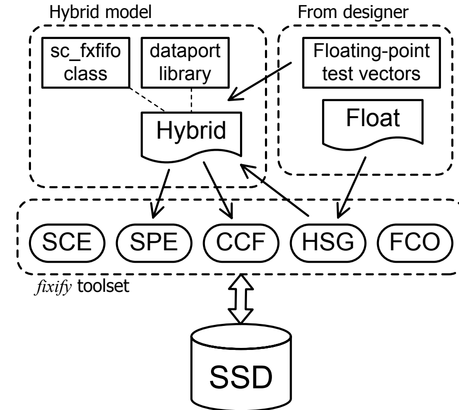

Figure 1. Structure of the *fixify* environment

In this paper, we present *fixify*, an environment for automated floating-point to fixed-point conversion, implemented as an integrated part of the OTIE described earlier. The rest of this paper is organized as follows. Section 2 describes the structure and organization of the *fixify* environment, followed by Section 3, describing the various optimization methods it offers to the designer. Results of processing a typical industrial DSP design with the *fixify* environment are presented in Section 4, including comparison of the optimization methods and the achievable control over the performance/cost trade-off. Finally, conclusions of the paper are drawn in Section 5.

2. The *fixify* Environment

The *fixify* environment is a tool chain developed within the OTIE to provide support for automated floating-point to fixed-point conversion. Hence, it is built around the core of the OTIE, the SSD, and employs a statistical approach to the optimization of fixed-point formats of all data channels in the system.

2.1. Structure of the Environment

A set of five tools forms the *fixify* environment, as shown in Figure 1: the Hybrid System Generator (HSG), the Corner Case Finder (CCF), the System Performance Estimator (SPE), the System Cost Estimator (SCE), and the Fixed-point Configuration Optimizer (FCO). The initial input by the designer is the pure floating-point design and the set of floating-point test vectors. The work flow as a design is processed by the *fixify* toolset is iterative and is shown in Figure 2.

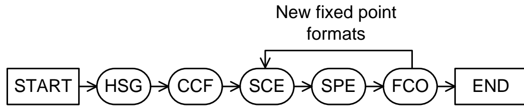

Figure 2. Typical flow through the *fixify* tool chain

2.2. Hybrid Model

The hybrid model of the system is used to estimate the degradation of its numeric performance by the introduction of fixed-point formats, relative to the pure floating-point model. This performance degradation is estimated through simulation of the hybrid model.

The key characteristic of the hybrid model is its ability to set any of the channels in the design to either floating-point or any fixed-point format. Hence, the entire hybrid system can then be used to simulate any combination of various fixed-point and floating-point formats in the design. It is also important to note that the hybrid model of the *fixify* environment is capable of fully transforming the numeric formats of any number of its channels at runtime, i.e. post compilation.

This is achieved through environment variables, which are monitored by the hybrid model and can be set by any of the tools (such as CCF or SPE), thus achieving finest-grain control over all numeric formats in the hybrid model at runtime. Aside from the sheer interface convenience this mechanism offers, its most important advantage is the dramatic time savings it offers to the *fixify* environment. Since the hybrid model can be reconfigured at runtime, during the optimization process, it needs to be compiled just once, rather than with every change of the fixed-point configuration.

The hybrid model is created by the HSG tool, as shown in Figure 1. Since the HSG tool operates directly on the system description provided by the designer, it is language-specific. Currently, the HSG tool supports SystemC [7] descriptions, but is built in a modular fashion, allowing for easy extensions to other system description languages. The *fixify* environment hence has access to and makes full use of all the fine controls over fixed-point formats available in the SystemC language. Also, the speedup in simulation offered by the limited-mantissa fixed-point formats of SystemC is automatically exploited whenever possible.

It is assumed that the original SystemC floating-point design, as supplied by the designer, is composed of **functional** and **structural** instances of modules, as is the usual design practice. Functional instances are those which contain one or more concurrent processes, which implement the functionality of the system. Structural instances are those which provide the structure and interconnections that make up the system.

The functional and structural instances are connected by data channels, altogether representing the design in the form of a graph, where instances are the nodes and data channels are the edges. The *fixify* environment optimizes the fixed-point formats of the data channels at this level of representation of the system, i.e. optimizes the formats of the edges of this graph. For systems that contain no feedback data channels, the graph representation of the system is a Directed Acyclic Graph (DAG).

The HSG tool uses unaltered functional instances of the original floating-point design to build the hybrid model. However, it modifies the structural instances to use `sc_fxfifo` channels, developed as part of the *fixify* environment. These channels implement the dynamic control over numeric formats in the system, as described earlier.

Also generated by the HSG tool is a testbench for the design. The automatically generated testbench contains interfaces to the supplied floating-point test vectors through the use of the `dataport` library, also written as part of the *fixify* environment. Compilation of the whole design is also automated by the HSG tool. After the hybrid model is compiled, it is delivered as an executable, for use by the rest of the *fixify* tool chain.

2.3. Corner Case

Introduction of fixed-point formats into a pure floating-point design creates degraded numeric performance, through the introduction of quantization and/or overflow errors. The optimization of fixed-point formats in the system is a trade-off between the reduced performance of the system and the reduced cost of the implementation through the reduction of fixed-point bitwidths.

At the extreme end of this trade-off lies the so-called **corner case fixed-point configuration**, defined as the set of minimum fixed-point formats for the entire system which produces no change to the numeric performance compared to the floating-point system and hence sets the maximum acceptable cost for the system implementation. Thus, the corner case is the ideal starting point for the optimization process, because all other fixed-point configurations which will be considered will have both worse performance and lower implementation cost.

It is the sole purpose of the CCF tool to find the corner case configuration for the hybrid system and place this information back into the SSD. The CCF tool first runs the hybrid system with all channels in floating-point format, and analyzes the bitwidth and sign format (signed/unsigned) requirements of each channel. Following this analysis, the CCF tool verifies the corner-case formats it found by running the hybrid system in this configuration and checking that the operation of the system is identical to that with floating-point formats. Once the corner case configuration is verified, the CCF tool stores this design refinement information into the SSD.

2.4. Cost and Performance Estimation

The two key metrics that are considered in the process of optimizing fixed-point formats in a design are the numeric performance of the system and its implementation cost. In the *fixify* environment, these metrics are estimated by the SPE and the SCE tools respectively.

Relative degradation of numeric performance of a system through the introduction of fixed-point formats is traditionally measured by the Signal-to-Quantization Noise Ratio (SQNR). This ratio is expressed as follows:

$$\text{SQNR} = 20 \times \log(E) \tag{1}$$

In the above expression, E is the relative error, or the relative difference between the original and quantized values (v_{orig} and v_{quan}, respectively):

$$E = \left| \frac{v_{orig} - v_{quan}}{v_{orig}} \right| \tag{2}$$

It is important to note that knowledge of the relative error alone is sufficient to determine the SQNR (through Equation 1). Hence, in the *fixify* environment, performance is expressed in terms of the relative error, E. This brings the advantage of reduced computational complexity, as well as easier handling of the case of no quantization error, where $E = 0$ and $\text{SQNR} = -\infty$.

The SPE tool estimates four performance metrics and places this refinement information back into the SSD. These metrics are E_{max_tot}, E_{avg_tot}, E_{max_ins}, and E_{avg_ins}. The maximum error metrics, E_{max_tot} and E_{max_ins}, correspond to SQNR_{min} (see Equation 1) and can thus be used to measure the worst-case quantization noise interference in the system. On the other hand, average error metrics, E_{avg_tot} and E_{avg_ins}, can be used in comparison to the maximum error metrics to determine the quality of a particular solution of the optimization process.

The inserted error measurements, E_{max_ins} and E_{avg_ins}, give us information on the exact error inserted in a particular channel due to its own fixed-point format. In other words, inserted error metrics do not include information on the quantization error that is already carried in the signal, inserted into the design by other channels, closer to the system inputs (in the graph representation of the system, as discussed in Section 2.2). The total error metrics, E_{max_tot}, and E_{avg_tot}, in contrast represent the total error experienced in the channel, i.e. the sum of the inserted error and the error inherited from channels closer to the system inputs.

All performance measures are used in conjunction with the global performance threshold, E_{limit}, set by the designer. This threshold corresponds to SQNR_{min} (see Equation 1), the highest acceptable quantization noise interference in the system.

Implementation cost of the system is also needed to perform the trade-off optimization that determines all the fixed-point formats in the system. The key requirement of the cost metric is to reflect the choice of bitwidth for all data channels in the system on its implementation cost. As such, it is independent of the actual implementation of each system component (hardware or software) and hence needs not be an absolute measure (such as seconds of runtime for software or gate count for hardware). However, it must be a good relative measure, so as to enable the optimization engine to choose the fixed-point configuration with smaller implementation cost than another. Hence, the SCE tool calculates the cost metric as a sum of all channel bitwidths and stores this refinement information back into the SSD.

Since neither the SPE nor the SCE tool interact directly with any system description code (neither the original floating-point description nor the hybrid model), both of these tools are independent of the description language used and require no modification to be used with languages other than SystemC.

2.5. Optimization Engine

The optimization engine of the *fixify* environment is implemented in the FCO tool. Since this tool interfaces only to the SSD (see Figure 1), it is also independent of the system description language used and requires no modification to be used with languages other than SystemC.

The FCO tool is a general framework for implementing optimization algorithms, providing interfaces to the performance and cost estimates in the SSD, as well as control over the numeric formats of all channels in the design. Hence, it is suitable for implementation of any number of various optimization methods. The optimization methods currently implemented in the FCO tool are described in Section 3.

3. Optimization Methods

The optimization engine of the *fixify* environment offers to the designer three different optimization algorithms: restricted-set full search, greedy search, and branch-and-bound.

3.1. Full Search

The basic optimization technique guaranteed to produce optimal results is the full search through the solution space. The most serious drawback of this approach is its long runtime, due to the fact that each possible solution needs to be considered separately.

For the floating-point to fixed-point conversion problem, the search space grows extremely quickly. If the number of data channels in the system is given as n and the corner-case bitwidth of the ith channel is given as w_i, the size of the search space, s, is given as:

$$s = \prod_{i=0}^{n-1} w_i \tag{3}$$

In other words, the total number of solutions considered by the full search algorithm grows exponentially with the number of channels in the system.

The range of w_i values is typically between 50 and 100, and even small designs contain 5 to 10 channels. Thus, the full search algorithm running on such a design needs to consider approximately $75^8 \approx 10^{15}$ possible solutions. To evaluate all the solutions in this vast search space, even with the simulation time of the hybrid model as short as say 0.5 seconds, the full search algorithm would take in excess of 15 million years to complete! Therefore, applying the full search algorithm to even the smallest of DSP designs is impractical.

3.2. Restricted-set Full Search

Although all variations on the basic full search algorithm require consideration of each possible solution in the search space and hence have complexity that grows exponentially with the number of channels in the system, it is possible to reduce the runtime of the full search algorithm down to the reasonable range of several minutes to several hours.

This can be achieved by reducing the set of possible bitwidths that can be assigned to a channel from N, the set of all positive integers, to a much smaller set, such as $\{16, 32, 64\}$ for example. Given that p_i is the number of possible bitwidths that can be applied to the ith channel, i.e. bitwidths in the restricted set that are smaller or equal to w_i, the search space can now be expressed as

$$s = \prod_{i=0}^{n-1} p_i \tag{4}$$

By restricting the set of possible bitwidths, the example mentioned earlier reduces in complexity from 10^{15} to just $3^8 = 6561$, which can be computed in less than an hour.

This approach is not only practical, but also highly suitable for optimization of designs that are aimed for implementation on DSP architectures, which will typically offer a restricted set of possible bitwidths, such as for example $\{16, 32, 40\}$ on the TI TMS320C62x architecture. It is also important to note that the restricted-set full search algorithm is guaranteed to find the optimal set of fixed-point formats for all data channels in the system.

3.3. Greedy Search

If the requirement of finding the guaranteed global optimum within the search space is relaxed, it is possible to implement optimization techniques that find near-optimal solutions in much shorter periods of time. One of these techniques is the greedy search algorithm. This algorithm is based on making steps through the search space, and when choosing the direction of the next step, always choosing the locally most favorable direction.

The greedy search algorithm of the *fixify* environment initially sorts the data channels in the design hierarchically, starting with the outputs, systematically working its way through the structure of the design, finishing with its inputs. Once the optimization sequence of data channels is determined, the bitwidth of each channel is optimized, keeping the formats of the rest of the channels fixed. This "upstream" sequence of optimization ensures minimal interference through inherited quantization error as each channel is optimized.

Therefore, each channel is optimized separately and only once, and the upper bound on the complexity of the greedy algorithm is given by:

$$s = \sum_{i=0}^{n-1} w_i \tag{5}$$

The number of required simulations thus grows linearly with n, the number of channels in the design. For the example design discussed earlier, the greedy search optimization has the complexity of only $75 \times 8 = 600$ (down from the original 10^{15}), and thus computes in just 5 minutes.

However, it must be noted that although greedy optimizations typically produce excellent results, they offer no guarantee of finding the optimum fixed-point configuration for the system.

3.4. Branch-and-Bound

Another optimization technique which produces faster results than the full search optimization is the branch-and-bound optimization. Branch-and-bound algorithms are based on the idea of representing the search space in the form of a tree structure and minimizing the time required to find the optimal solution by intelligent traversal of this tree.

Indeed, branch-and-bound algorithms systematically exclude sub-branches of the tree relative to their current location, when they can infer that these sub-branches cannot contain the optimum solution. These algorithms thus make overwhelming savings in complexity by being able to exclude parts of the search space through their inherent knowledge of the problem. In other words, branch-and-bound algorithms find the same global optimum full search algorithms do, but contain intelligent and problem-specific mechanisms to dramatically shrink the search space. In particular, the non-decreasing (isotonic) nature of the cost function with resprect to channel bitwidth is exploited, as will be seen later.

The branch-and-bound optimization technique of the *fixify* environment starts off by producing the optimization sequence of data channels, in the same way greedy search optimization does. Data channels are then optimized individually, again in the same way greedy optimization does, but having found the minimal bitwidth of each channel this way, instead of proceeding to the next channel in the sequence, branch-and-bound optimization does not discard the rest of the solutions in the current subtree of the solution space, but proceeds to look through it, looking for the global optimum.

Once the bitwidth of the current channel is minimized, the branch-and-bound algorithm assembles the list of data channels in the design that are influenced by the current channel, i.e. are found "downstream" in the dataflow graph of the system. Following this, all possible combinations of tightening the current channel further and relaxing any combination of the subtree channels is explored.

However, rather than considering every possible combination like the full search algorithm would, the branch-and-bound algorithm takes further shortcuts during this exploration process, i.e. excludes parts of the solution subtree. Any possible solutions which increase the implementation cost are not considered. This removes a very significant part of the search space. Also, the iterative tightening of the current channel and relaxation of the subtree channels does not go on beyond the point where the inserted maximal error (E_{max_ins}) in the current channel exceeds the global

threshold (E_{limit}) set by the designer, which also removes a significant part of the search space.

While the worst-case complexity of the branch-and-bound algorithm is equal to that of full search optimization (see Equation 3) and grows exponentially with the number of channels in the system, this optimization typically executes orders of magnitude faster, due to the minimization of the search space. On the other hand, branch-and-bound optimization is still guaranteed to find the optimum fixed-point configuration for the system, just like full search optimization.

However, this guarantee applies only to systems whose dataflow graphs can be represented as DAGs, i.e. systems which do not include feedback signals. Systems which contain feedback cycles in the dataflow graph cannot be resolved to an optimization sequence where all the signals in the subtree of the current channel do not influence the current channel itself. For such systems, all cycles in the dataflow graph are broken and the systems are then analyzed as described above, but results cannot be guaranteed to be optimal.

4. Results

In order to demonstrate the viability of employing the *fixify* environment on realistic DSP designs, as well as investigate the relative performance of its three optimization techniques, an industrial DSP design was chosen as a benchmark and processed by the *fixify* environment.

4.1. Benchmark DSP Design

The design used as a benchmark in this work is the slot synchronisation part of the cell searcher module, which is a standard part of the UMTS receiver. Its function is to detect the start of the slot transmitted by the node B (base station) by correlating the received signals with the primary synchronisation code and hence achieve slot synchronisation.

The cell searcher module was described in SystemC 2.0 and its structure broken down into four sub-modules, connected by eight data channels. Also supplied to the *fixify* environment was a set of floating-point test patterns, corresponding to a full frame of received data, i.e. 15 slots of 2560 chips, or a total of 38400 values.

Automated conversion was performed by the *fixify* environment on the cell searcher design, using four different optimization methods. Firstly, restricted-set full search with the possible bitwidth set of $\{16, 64\}$ was used. The second optimization method was again restricted-set full search, but with the set of possible bitwidths being $\{16, 32, 64\}$. The third and fourth optimization methods were greedy and branch-and-bound, respectively. Each of the four optimization methods was employed with eight different error limits, corresponding to the SQNR$_{min}$ range of [5dB, 40dB].

From the results of these automated conversions, conclusions can be drawn about the relative performance of each optimization

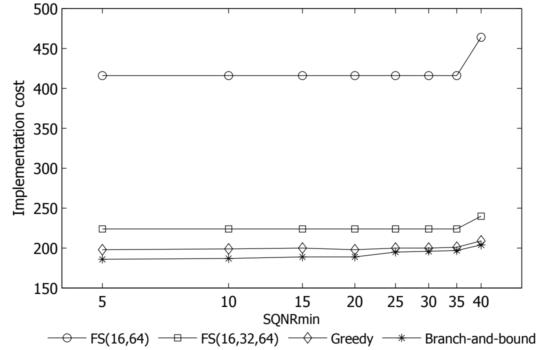

Figure 3. Optimization results

method, both in terms of the quality of its results and its runtime performance.

4.2. Quality of Results

Since the *fixify* environment is aimed at replacing the designer in making the trade-off between implementation cost and numeric performance of the fixed-point implementation, the performance of the environment can thus be measured through the achieved minimization of system implementation cost for each given lower bound on numeric performance (SQNR$_{min}$). The system implementation cost obtained by each optimization technique is shown in Figure 3.

It can immediately be noted that both instances of the restricted-set full search algorithm obtain relatively high system implementation costs. Also, these algorithms do not reduce the system implementation cost over most of the SQNR$_{min}$ range. Both of these effects stem directly from the nature of the restricted-set full search algorithm.

Since the number of possible bitwidths this algorithm considers for each channel is severely limited (to just two or three bitwidths in this experiment), the algorithm cannot exercise fine control over the resulting implementation cost. In other words, only large steps within the search space are available to this optimization method, leading to a situation where even a significant change in the allowed performance degradation (such as between 30dB and 5dB in Figure 3) still does not open a possibility of taking another large step in the search space and making the correspondingly large reduction in system cost.

It is also important to note that with increased freedom through a larger set of possible bitwidths, the restricted-set full search algorithm manages to significantly improve the quality of its results.

It is evident from Figure 3 that both the greedy search and the branch-and-bound algorithms, both of which consider solutions in the continuous search space (i.e. for which each integer bitwidth is a possible solution), obtain superior results to the restricted-set full search algorithm. It can also be noted that the branch-and-bound algorithm performs better than the greedy search, by a margin of 2% - 6%, due to the sub-tree relaxation technique, described in

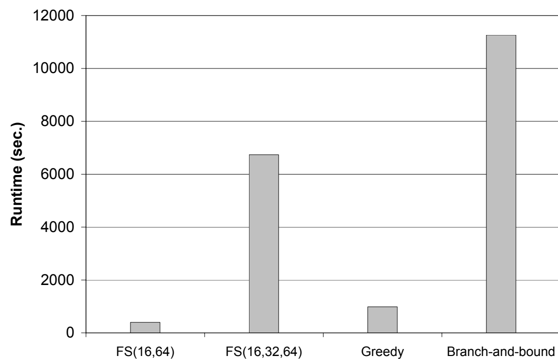

Figure 4. Runtime performance results

Section 3. Both algorithms are able to keep fine-grained control over the cost - performance trade-off, achieving a total cost reduction of 5% to 10% over the examined SQNR_{min} range.

When the optimization results for each channel in the design found by the restricted-set full search algorithms and those found by the continuous search space methods (greedy and branch-and-bound) are compared, it can be noted that they correspond exactly. In other words, for each channel, when the continuous search space results are quantized upward to the next allowed bitwidth for one of the restriced-set methods, they are found to be equal to the result found by the restricted-set method itself.

4.3. Runtime Performance

Practicality of using the *fixify* environment as part of a complete design flow of a DSP design does not only depend on the quality of its results, but is also strongly influenced by the required runtime. The average required runtimes for each of the optimization methods is shown in Figure 4.

As described in Section 3, the required runtime of the restricted-set full search algorithm increases polynomially with the size of the set of possible bitwidths, and this can be seen clearly in Figure 4, where the size of the set of possible bitwidths increased from two to three.

Also worth noting is that, although the branch-and-bound algorithm produces superior results to the greedy algorithm by employing the sub-tree relaxation technique, this in turn introduces a significant overhead in the required runtime. In other words, the 2% to 6% improvement in the quality of results is offset by a more than ten-fold increase in the required runtime.

5. Conclusions

The *fixify* environment has been successfully applied to performing a fully automated floating-point to fixed-point conversion, thereby completely replacing the designer's manual effort. Furthermore, it has been applied to an industrial DSP design, with runtimes in the order of minutes, thus proving practicality of its application.

For designs that are to be mapped to software running on a DSP core, restricted-set full search is the best choice of optimization technique, since it offers guaranteed optimal results and optimizes the design directly to the set of fixed-point bitwidths that are native to the DSP core in question. For custom hardware implementations, the best choice of optimization option is the branch-and-bound algorithm, offering guaranteed optimal results. However, for high-complexity designs with relatively long simulation times, the greedy search algorithm is an excellent alternative, offering significantly reduced optimization runtimes, with little sacrifice in the quality of results.

References

[1] B. Knerr and M. Holzer and M. Rupp. HW/SW Partitioning Using High Level Metrics. In *International Conference on Computer, Communication and Control Technologies CCCT'04*, volume VIII, pages 33–38, Austin, Texas, Aug 2004.

[2] C. F. Fang, R. A. Rutenbar, and T. Chen. Fast, Accurate Static Analysis for Fixed-point Finite Precision Effects in DSP Designs. In *International Conference on Computer Aided Design*, San Jose, Nov 2003.

[3] International SEMATECH. International Technology Roadmap for Semiconductors, 1999.

[4] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty. Model-Integrated Development of Embedded Software. In *Proceedings of the IEEE*, volume 91, pages 145–164, Jan 2003.

[5] H. Keding, M. Willems, M. Coors, and H. Meyr. FRIDGE: A Fixed-Point Design and Simulation Environment. In *Design, Automation and Test In Europe DATE'98*, Feb 1998.

[6] S. Kim, K. Kum, and W. Sung. Fixed-Point Optimization Utility for C and C++ Based Digital Signal Processing Programs. *IEEE Transactions on Circuits and Systems-II: Analog and Digital Signal Processing*, 45(11):1455–1464, Nov 1998.

[7] Open SystemC Initiative. www.systemc.org.

[8] P. Belanović, M. Holzer, D. Mičušík, and M. Rupp. Design Methodology of Signal Processing Algorithms in Wireless Systems. In *International Conference on Computer, Communication and Control Technologies CCCT'03*, pages 288–291, Jul 2003.

[9] P. Belanović, M. Holzer, B. Knerr, M. Rupp, and G. Sauzon. Automatic Generation of Virtual Prototypes. In *International Workshop on Rapid System Prototyping RSP'04*, pages 114–118, Geneva, Switzerland, Jun 2004.

[10] M. Rupp, A. Burg, and E. Beck. Rapid Prototyping for Wireless Designs: the Five-Ones Approach. *Signal Processing Europe 2003*, Jun 2003.

[11] C. Shi and R. W. Brodersen. An Automated Floating-point to Fixed-point Conversion Methodology. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, pages 529–532, Apr 2003.

[12] R. Subramanian. Shannon vs. Moore: Driving the Evolution of Signal Processing Platforms in Wireless Communications. In *IEEE Workshop on Signal Processing Systems SIPS'02*, Oct 2002.